Fall 12-14-2010

Data Aggregation through Web Service Composition in Smart Camera Networks

Jayampathi S. Rajapaksage
Georgia State University

DATA AGGREGATION THROUGH WEB SERVICE COMPOSITION IN SMART CAMERA NETWORKS

by

JAYAMPATHI RAJAPAKSAGE

Under the Direction of Sushil K. Prasad

ABSTRACT

Distributed Smart Camera (DSC) networks are power constrained real-time distributed embedded systems that perform computer vision using multiple cameras. Providing data aggregation techniques that is critical for running complex image processing algorithms on DSCs is a challenging task due to complexity of video and image data. Providing highly desirable SQL APIs for sophisticated query processing in DSC networks is also challenging for similar reasons. Research on DSCs to date have not addressed the above two problems. In this thesis, we develop a novel SOA based middleware framework on a DSC network that uses Distributed OSGi to expose DSC network services as web services. We also develop a novel web service composition scheme that aid in data aggregation and a SQL query interface for DSC networks that allow sophisticated query processing. We validate our service orchestration concept for data aggregation by providing query primitive for face detection in smart camera network.

INDEX WORDS: Smart camera networks, OSGi, Distributed OSGi, Web service composition, SQL query language primitives, Face detection, Data aggregation

DATA AGGREGATION THROUGH WEB SERVICE COMPOSITION IN SMART CAMERA NETWORKS

by

JAYAMPATHI RAJAPAKSAGE

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Sciences

Georgia State University

2010

DATA AGGREGATION THROUGH WEB SERVICE COMPOSITION IN SMART CAMERA NETWORKS

by

JAYAMPATHI RAJAPAKSAGE

DEDICATION

To my wife and family for their unconditional support and motivation.

ACKNOWLEDGEMENTS

This thesis is the result of research carried out over a period of two years. During this period, many people supported my work and helped me to bring it to a successful conclusion, and here I would like to express my gratitude.

First, I would like to express my gratitude to my advisor, Dr. Sushil K. Prasad, for his support and invaluable guidance throughout my study. His knowledge, perceptiveness, and innovative ideas have guided me throughout my graduate study. I also present my words of gratitude to the other members of my thesis committee, Dr. Rajshekhar Sunderraman, and Dr. WenZhan Song, for their advice and their valuable time spent in reviewing the material. I especially want to express my sincere gratitude to Dr. Sunderraman for all his help, advice and support during my graduate studies. I deeply appreciate his valuable advice and support on both professional and personal matters. I also would like to extend my appreciation to my colleagues in DiMoS research group for all the support and ideas and to everyone who offered me academic advice and moral support throughout my graduate studies.

Finally, I would like to express my gratitude to my family, especially my wife, for their unconditional support and outstanding belief in my success. Without their support, this research project would not have been possible. Their continuous support played an essential role in helping to evolve my ideas and improve the quality of this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Motivation

Distributed Smart Camera (DSC) networks are real-time distributed embedded systems that perform computer vision using multiple cameras. A Smart camera is a powerful sensor that can process complex information such as image and video streams. Similar to Wireless Sensor Networks (WSN), the major requirements of DSC networks include low power consumption and advanced communication capabilities. However, while resource constraints of embedded smart cameras are important, DSC networks are not as resource limited as WSNs.

Smart camera applications have gained a lot of attention in the research community lately. Applications for distributed smart camera networks are interactive, dynamic, stream based, computationally demanding and need real time or near real time guarantees. Designing, implementing, and deploying applications for DSC networks is a complex and challenging task due to the multiple and heterogeneous nature of DSC networks.

Being a resource limited environment, it is crucial in a DSC network that both bandwidth utilization and energy consumption is minimized at node-level processing. Also, keeping human eye on all the video streams produced by the cameras is not possible, as massive amounts of sensed video streams need to be properly handled. This demands for automated object analysis algorithms running on top of the DSC networks. These algorithms should enable situation awareness and deployment of resources (e.g. Place more cops in highly active areas) in a cost effective manner.

Data aggregation is one of the basic functionalities that is performed in a distributed environment. Running complex image processing algorithms (e.g. object monitoring, object recognition, motion detection etc.) on top of DSC networks, require the aid of data aggregation techniques. Numerous techniques for data aggregation have been proposed over the past recent years for WSNs [1, 39]. Conventional WSNs only aggregates simple data such as humidity and temperature. DSC networks, on the other hand, require aggregation of more complex data such as videos and images. To our knowledge, this problem has not been studied by the research community to date, and still remains to be an open problem.

High-level query languages are attractive interfaces for WSNs and DSC networks, potentially relieving application programmers from the burdens of distributed and embedded programming. Researchers have often highlighted the benefits of query like interfaces for such networks. There have been efforts in the past to develop query like interfaces for sensor networks [1, 39, 40]. SQL APIs for DSC networks needs to provide queries that allow for complex image and video analysis functions to the user. To best of our knowledge, none has tackled the problem of providing a query like interface for data aggregation in DSC networks before.

Service Oriented Architecture (SOA) has been proven to support increasing complexity and heterogeneity of nowadays' information systems. Handling structured data formats used in web services is a key challenge in energy and bandwidth limited DSC nodes. By enabling web services in such environments we can expose functionality and data produced by the smart camera nodes in a highly interoperable manner.

This thesis will address these important issues together by developing a middleware framework for DSC networks which allows for efficient implementation of distributed object monitoring algorithms.

Rest of this introduction is organized as follows: Section 1.2 describes the problem statement of the thesis. Section 1.3 lists the contributions made in this thesis. Finally, section 1.5 provides an overview of the thesis organization.

1.2 Problem Statement

This thesis focuses on following problems:

Problem 1: Data Aggregation in DSC Networks

Data aggregation in sensor networks is a well studied problem. Sensor networks aggregate simple data such as temperature, light, vibrations etc. Aggregating complex data types like images and videos in DSC networks however is not a well studied problem by the research community. It is very important and useful to enable data aggregation in such networks for situation awareness and efficient resource allocation. In fact, complex data aggregation itself is a new and evolving research topic.

Problem 2: Scalable Operation of DSC Networks

A highly scalable DSC network should provide easy extension of the system with new functionalities and services. Providing scalability in DSC network operations is a major challenge, as this requires the support of platform independent, language independent and most importantly vendor independent operations. Many sensor networks lack scalability in this respect, as their programming languages and environments are vendor specific. We plan to address this issue by providing a web service enabled DSC network.

Problem 3: High Level Query Language Primitives for Sophisticated Image Data Analysis

Similar to sensor networks, DSC network programming mixes complexities of both distributed and embedded system design. This problem is further amplified by limited physical resources. Providing high level query language primitives or SQL APIs for such networks relieves the programmer from the burden of distributed and embedded programming. To this date, no SQL APIs exists for data aggregation in DSC networks, to the best of our knowledge.

Adapting SQL query interfaces provided for traditional distributed computer systems in a DSC network is infeasible for two major reasons: First, SQL APIs for traditional distributed computer systems were not designed to handle low power consumption requirements of DSC networks. Second, services provided by such APIs are too generic to meet the functional requirements of application specific networks such as DSC networks. Adapting SQL APIs developed for sensor networks in DSC network environments is also infeasible as these SQL APIs are developed to support applications specific to sensor networks. Moreover, the attempts to extend proposed SQL APIs for sensor networks with sophisticated queries seems to have reached a dead end due to usage of vendor specific platforms, languages and high learning curve of low level programming.

Providing high level SQL APIs for sophisticated data analysis in DSC networks remains an unstudied problem to this end. In our thesis work, we provide high level SQL primitives that allow users to issue queries that perform sophisticated image analysis functionalities. We propose a method to map these SQL queries to web service compositions to carry out the intended tasks.

1.3 Research Contributions

In this thesis work we develop a highly scalable SOA enabled DSC network architecture that provides high level SQL primitives for sophisticated image data analysis. More precisely, our research contributions focus on the following:

Contribution 1: Provide Scalable Operation by Enabling Web Services in DSC Networks

Scalability is one of the most desired qualities in a DSC network. The unscalability of DSC networks could arise due to a variety of reasons including vendor specific languages and platforms used in the DSC environment. Exposing camera functionalities as web services, therefore, is an excellent way to provide high scalability to DSC networks.

To provide such a web service enabled environment, we implement our DSC network on top of Distributed Open Source Gateway initiative (Distributed OSGi) framework [35]. OSGi [34] is a dynamic module system for java that is fast gaining momentum as a framework for developing and deploying modular reusable java programs which is mostly being used in embedded systems. In this work, we use OSGi to allow camera nodes to publish, discover and consume their functionalities as local services. Distributed OSGi enables these services to be published as web services by providing the distribution capability to the DSC network. Enabling web service on top of distributed OSGi in DSC network this way exposes the functionality and data provided by the camera nodes in highly interoperable manner. OSGi also allows the developer to implement more sophisticated and highly scalable data analysis algorithms by introducing new services to the DSC network and by extending existing services in the network. More details on OSGi framework and how we utilize it to provide a web service enabled DSC environment can be found in sections 2.5 and 3.1.

Contribution 2: Provide Data Aggregation through Web Service Composition

We introduce a novel smart camera architecture for data aggregation through web service composition in DSC networks on top of distributed OSGi framework. Since functionalities offered by the DSC network are exposed as a web services through Distributed OSGi, a data aggregation problem (e.g. finding maximum number of faces seen in a given region in a given period of time) can be transformed into a web service composition problem. For example, in an simplified scenario where aggregation is the maximum number of faces detected, the composition would involve elementary web service such as face detection web service, region identification web service and maximum aggregate finding web services etc. We present a process model and dynamic service realization and binding method that aid this service composition.

Contribution 3: Provide SQL API for Image Data Analysis

Many potential users of sensor networks are not computer scientists. In order for these users to develop new applications on sensor networks, high-level languages and corresponding execution environments are desirable. A query-based approach can be a good general-purpose platform for application development for such users. Therefore, we propose a high level query API with SQL like language primitives that allow the user to perform sophisticated data analysis such as object detection in the DSC network.

Contribution 4: Efficient Resource Management in DSC Networks Using OSGi Life Cycle

Resource management is one of the important tasks in DSC networks. We propose a method to manage resources in a DSC environment using OSGi life cycle. OSGi contains *bundles* which are behavioral components that offer libraries statically and services dynamically. In a web service environment they can be thought of as a single web service or a collection of web services. Bundles provide a component base environment and offer component life cycle services such as installing, starting, stopping, and uninstalling. Therefore OSGi life cycle component can effectively be used for resource management in the DSC network. Starting necessary bundles when a service is requested by the user, and stopping bundles when the service completes in our DSC network will model dynamic service loading and unloading behaviour.

1.4 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 provides an overview of the related technologies to this thesis and a careful evaluation of their use. Chapter 3 describes the thesis requirements and the architecture proposed to support web service enablement, SQL API for sophisticated data analysis and data aggregation, and web service composition. This chapter provides an architectural view of the whole system. Chapter 4 walks through the system we developed to support the proposal and present and discuss our results of our deployment. Lastly, in Chapter 5, we present our concluding remarks, the limitations and potential enhancements and additional research ideas.

2 BACKGROUND AND RELATED WORK

This chapter focuses on providing the reader a high-level view of the following concepts: DSC networks, data aggregation, web service composition, query processing APIs for such networks, and distributed OSGi. Note that this is a discussion of works more closely related to our work presented in this thesis. In addition to work done in DSC networks, many state of the art techniques described in this chapter are related to sensor networks as this is the most similar type of network to DSC networks we can find in terms of resource availability requirement and functional and non functional requirements. We also specifically select and discuss two related works, TinyDB [1] and MORE [72] that is aligned more with the goals of this thesis.

2.1 Distributed Smart Camera Networks

Smart camera networks are real-time distributed embedded systems that perform computer vision using multiple cameras [5]. This new approach has emerged thanks to a confluence of simultaneous advances in four key disciplines: computer vision, image sensors, embedded computing, and sensor networks. There has been a dramatic progress in smart camera research and development over the recent years. Cameras are no longer boxes and no longer take pictures. The fundamental purpose of a smart camera is to analyze a scene and report items and activities of interest to the user. The basic output of a smart camera is not an image. Multiple smart cameras can be utilized to cover a large space and perform smart camera operations by coordinating with each other. Such a network is called a *Distributed Smart Camera network* or a *DSC network*. DSC network are typically wired. Wireless DSC network also exist, but not to the extent of wired DSC networks.

Distributed smart cameras are used in a variety of application arrears including law enforcement, security machine vision, medicine, and entertainment. Data analysis in DSC networks generally involves aggregation of image data from multiple cameras. The cameras need to cooperate with each other due to the complex geometric relationships between subjects of interest in the considered scene. This complex geometric relationship may come in the form of the trajectory of a moving object, poses of humans etc. resulting in dynamic group of cameras interacting over a given period of time. Pulling all of the videos from a large number of cameras to a central server is expensive and inherently unscalable making server-based architectures a bad choice for a DSC network.

Distributed Smart Camera Networks vs. Sensor Networks

A DSC network is significantly different from other well known network types such as traditional distributed computer systems [20] and sensor networks [21] in terms of processing capacity, communication capability and applications. Therefore, data aggregation techniques, object monitoring algorithms and SQL like query interfaces available for traditional computer networks or sensor networks cannot directly be adopted for DSC networks. The smart camera nodes in a DSC network are more powerful in function and are of smaller size. They can acquire and process not only simple information like temperature and humidity, but also complex information like videos and images. As a result, in addition to basic characteristics of WSNs, DSC networks have access to massive amounts and sorts of information acquired from the environment.

2.1.1 Processing in Distributed Smart Camera Networks

Processing in DSC networks can often be achieved by sending data to a centralized node to process the data. However, this is quite impractical in most situations due to the scalability requirement of a distributed network. Therefore, distributed processing is often required to meet this goal.

(i) Centralized Processing

Optimal image processing algorithms can be devised if all the images can be sent to and processed at a given central node. However, transmitting real-time video streams to a central node is infeasible not only due to the large bandwidth consumed by these video streams but also due to energy consumed for computation of large amount of data at a given smart camera node. Having smart cameras with onboard processing unit in your network in such cases however, is beneficial in achieving these hard to achieve goals.

When images are made available to a smart camera node, the vision processing algorithms at the node can employ numerous techniques to detect, recognize and track objects in the images. Two popular approaches used in such vision processing algorithms are *top-down approach* and *bottom-up approach*. Typical for the top-down approach is to build a model that is fitted to the images, for example fitting 3D human models to images. The most popular however is the bottom up approach where various features are extracted from images which are then combined within an image and then across images from different camera views. Many bottom-up approaches can naturally be mapped to a DSC network [5].

(ii) Distributed Processing

This is the most desirable form of processing in DSC networks. In an ideal distributed setting, there exists a fully decentralized vision processing algorithm that computes and disseminates aggregates of the data with minimal processing and communication requirements and good fault tolerance.

However, tt is highly likely that hybrid processing will lead to the best practical solution, where cameras form groups to combine information in a centralized way. Further data processing can also be distributed across groups [5].

2.1.2 Smart Camera Architecture

The smart camera architecture is generally similar to that of a general sensor node. However, a smart camera also contains necessary hardware and software capabilities to processes visual information. The main hardware module of a smart camera includes the sensor, processor and the communication unit. Figure 2.1 depicts the generic architecture of a smart camera.

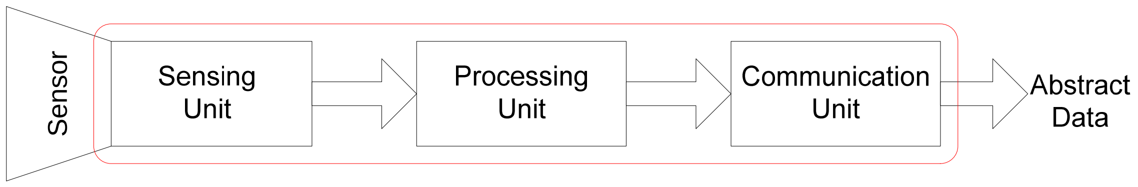


Figure 2:1 Generic Architecture of a Smart Camera

(i) Sensing Unit

The fundamental purpose of the sensing unit is to transform light into digital signals and perform basic image enhancement tasks such as white balance, contrast, and gamma correction. For the sensor module, any kind of image sensor can be used based on the application. Low end image sensors are more suitable for DSC networks due to their smaller size and low cost. A smart camera may be equipped with a number of camera sensor modules for numerous purposes such as depth estimation or using different resolution for different tasks.

(ii) Processing Unit

Image data from the sensor module is processed by the processing unit. The onboard processor also controls the sensor parameters and communication. Choosing an appropriate processor is difficult because the required amount of processing power largely depends on the application. Media Processors and Vision/Image processors are two of the processors mostly used in the processor module.

(iii) Communication Unit

The communication module is used to handle the communication between smart camera nodes and/or a central processing unit. For DSC networks, wireless communication is preferred over wired communication for many practical reasons. One of the main issues arising from this preference however is the decision of what information needs to be transmitted to other cameras or to a central node to maximize the network lifetime. This can range from sending real-time video streams to sending just events detected by the camera. The better choice is to invest more on processing in node itself and send only events detected.

2.1.3 Types of Smart Cameras

There are three types of smart cameras identified in the literature [5]:

(i) Single Smart Cameras(SSCs)

SSCs integrate the sensing with embedded on-camera processing. By doing so, the SSCc are able to perform various vision tasks onboard and deliver abstracted data from the observed scene.

(ii) Distributed Smart Cameras(DSCs)

DSCs introduce distribution and collaboration to smart cameras, resulting in a network of cameras with distributed sensing and processing. Distributed smart cameras therefore collaboratively solve tasks such as multi-camera surveillance and tracking by exchanging abstracted features. Table 1.1 [5] lists some example DSCs.

(iii) Pervasive Smart Cameras(PSCc)

PSCs integrate adaptivity and autonomy to DSC networks. The ultimate vision of PSCs is to provide a service-oriented network that is easy to deploy and operate, adapts to changes in the environment, and provides various customized services to users.

The main focus of our thesis work is DSCs. The heterogeneity of DSC networks may come from different capabilities such as sensing, processing and communication. This inherent heterogeneity in DSC networks allows for dynamic adaptation to environment during operation. The optimization goals for DSC network are again many fold depending in the application. Few such optimization criteria include energy, time and communication bandwidth. A freely moving camera not only poses challenges for calibration and background elimination. Connected by some wireless links, it can also change the topology of the overall network. Communication links to some nodes may drop; new links to other nodes may need to be established.

Table 2:1 Examples of Distributed Smart Camera system

System	**Platform Capabilities**			**Application**
	Sensor	**CPU**	**Communication**	
Distributed SmartCam [78]	VGA	ARM and multiple DSPs	100-Mbps Ethernet, GPRS	Local image analysis, Cooperative tracking
BlueLYNX [79]	VGA	PowerPC, 64-MB RAM	Fast Ethernet	Local image preprocessing Central reasoning
GestureCam [80]	CMOS, 320×240 (max.1280×1024)	Xilinx Virtex-II FPGA, custom logic plus PowerPC core	Fast Ethernet	Local image analysis, No collaboration
NICTA Smart Camera [81]	CMOS 2592×1944	Xilinx XC3S5000 FPGA; microBlaze core	GigE vision interface	Local image analysis, No collaboration

2.2 Data Aggregation

One important enabling technology for DSC networks and WSNs in general is data aggregation, which is essential for the network to be reusable and cost-efficient. The data aggregation can be divided into three parts: data acquisition, data transmission and data processing. These three steps have a close contact with the efficiency and quality of utilizing the wealth of data resource.

Data aggregation attempts to collect the most critical data from the camera nodes and make them available to the sink in an energy efficient manner with minimum data latency. Data latency is an important factor in real time systems such as DSC networks. The database community has proposed a number of distributed and push-down based approaches for aggregates in database systems [23, 24], but these universally assume a well-connected, low-loss topology that makes these approaches inapplicable in sensor networks. None of these systems present techniques for loss tolerance or power sensitivity.

Data gathering in sensor networks (and DSC networks) is defined as the systematic collection of sensed data from multiple sensors to be eventually transmitted to the base station for processing. Since sensor nodes are energy constrained, it is inefficient for all the sensors to transmit the data directly to the base station. Data generated from neighboring sensors is often redundant and highly correlated. In addition, the amount of data generated in large sensor networks is usually enormous for the base station to process. Data aggregation allows for combining data into high-quality information at the sensors or intermediate nodes thus reducing the number of packets transmitted to the base station resulting in conservation of energy and bandwidth.

Data latency is important in many applications such as environment monitoring, where the freshness of data is also an important factor. It is critical to develop energy-efficient data aggregation algorithms so that network lifetime is enhanced and data aggregation results in minimum latency in data delivery.

2.2.1 Data Aggregation Based on Network Architecture

The network architecture plays an important role in data aggregation algorithm performance. Much work has been done in the area of sensor networks for data aggregation. We will discuss and analyze several such data aggregation techniques used in *flat network* and *hierarchical network* architectures for sensor networks.

(i) Data Aggregation in Flat Networks

In *flat networks*, each sensor node plays the same role and is equipped with approximately the same battery power. In such networks, data aggregation is accomplished by *data centric routing* where the sink usually transmits a query message to the sensors. SPIN [41] is one such sensor protocol that use *push based diffusion scheme* where the sources are the active participants which initiate the diffusion and sinks simply respond to the sources. For successful data negotiation, SPIN describes its observed data as metadata and this data transmission takes place after a resource (e.g. energy) poll between sources for overall maximization of resource consumption in the network. According to experimental studies, SPIN performs almost identically to flooding but incurs 3.5 times less energy. Direct Diffusion [42] is yet another sensor protocol which falls into the category of *two-phase pull diffusion scheme* where data is acquired at the sensor based on data-centric routing. Direct diffusion is achieved by using data-driven local rules. Sink initially broadcasts a request and source sensors with matching data, sets up a gradient that specifies the data rate and the direction to send the data. The intermediate nodes are capable of caching and transforming the data. After the receipt of the low-data-rate events, the sink reinforces one neighbor that provides higher quality data. The average dissipated energy in directed diffusion is only 60 percent of the omniscient multicast scheme according to simulations. Two-phase pull diffusion however results in large overhead if there are many sources and sinks. This can be overcome by skipping the flooding process of direct diffusion as proposed by Krishnamachari in [43].

One of the major disadvantages in flat networks is that they can results in excessive communication and computation at the sink node. This incurs faster consumption of battery power minimizing the network lifetime. Therefore, data aggregation protocols deployed in flat networks are unscalable and not energy efficient

(ii) Data Aggregation in Hierarchical Networks

In *hierarchical networks*, the above mentioned disadvantages in a flat network are eliminated. The overhead of computation at the sink is reduced in hierarchical aggregation method by performing some data aggregation at special nodes in the hierarchy from sources to the sink. This also reduces the number of messages transmitted to the sink reducing the overall battery power consumption in the network. One of the most popular forms of hierarchical networks is *cluster-based networks* where data is transmitted from sensors to a local aggregator and these local aggregators in turn send these partially aggregated data to the sink. LEACH [44] is one such protocol where sensors are organized into clusters for data aggregation. A cluster head is selected from each cluster to perform the local aggregations and send these results to the sink. This data aggregation performed periodically at the cluster heads. While LEACH improves the network lifetime by preserving energy, it has some limitations. For example, it assumes each node has the capability to act as the cluster head which is not realistic for a network with heterogeneous sensors with power constraints. HEED [45] overcomes these issues in LEACH by assuming multiple levels of power in sensors. The main goal of HEED is to maximize network lifetime by efficient cluster formation. The cluster heads in HEED are selected based on the residual energy at each sensor node and node proximity to its neighbors or node degree. Data aggregation based on cluster-based networks however, often suffers from energy waste due to long distance cluster heads from the sensor nodes in the cluster. *Chain-based data aggregation* allows sensors to transmit only to close neighbors eliminating the above problem. Lindsey *et al.* [12] presented a chain-based data-aggregation protocol called Power-Efficient Data-Gathering Protocol for Sensor Information Systems (PEGASIS). Here, nodes are organized into a hierarchical chain using some greedy algorithm or centrally by the sink. The chain information is initiated by the node in the chain furthest to the sink and each intermediate node in the chain aggregates its own data with data received from neighbor in the chain and sends to its other neighbor closer to the sink. Eventually, the aggregated data reaches the sink. The PEGASIS protocol has considerable energy savings compared to LEACH. The effectiveness of the chain-based data aggregation however, largely depends on the construction of the energy efficient chain. *Tree-based data aggregation* is yet another popular hierarchical data aggregation technique. Here, the sensors are organized into a tree where the root is the sink. Data aggregation is initiated at leaf nodes and when a parent node receives data from its children it fuse its data with received data and send to its parent. Eventually the root or the sink will receive the aggregated data. EA-DAT proposed in [47] maintains such a data aggregation tree. However, in this case, the data aggregation is initiated by the sink or the root by broadcasting a control message which results in nodes organizing into a tree and sending aggregated data towards the root.

In out thesis work, we assume a hierarchical DSC network due to their numerous advantages. Table 2.2 shows a comparison in data aggregation for flat networks vs. hierarchical networks [22].

There has not been much research done in data aggregation in DSC networks. Data aggregation in DSC networks is more complicated than that of sensor networks due to the complex data that need to be processed and aggregated. However there has been research attempts to perform sensor fusion [48, 49] where many other types of sensors such as infrared cameras, audio sensors are integrated with visual camera sensors to provide more reliable and robust data with reduced uncertainty and ambiguity.

Table 2:2 Data Aggregation in Hierarchical Networks versus Flat Networks

Hierarchical networks	Flat networks
Data aggregation performed by cluster heads or a leader node.	Data aggregation is performed by different nodes along the multi-hop path.
Overhead involved in cluster or chain formation throughout the network.	Data aggregation routes are formed only in regions that have data for transmission.
Even if one cluster head fails, the network may still be operational.	The failure of sink node may result in the breakdown of entire net-work.
Lower latency is involved since sensor nodes perform short-range transmissions to the cluster head.	Higher latency is involved in data transmission to the sink via a multi-hop path.
Routing structure is simple but not necessarily optimal.	Optimal routing can be guaranteed with additional overhead.
Node heterogeneity can be exploited by assigning high energy nodes as cluster heads.	Does not utilize node heterogeneity for improving energy efficiency.

2.3 Web Service Composition

2.3.1 Service Oriented Architecture (SOA)

SOA refers to a style of building reliable distributed systems that deliver functionality as *services*, with the additional emphasis on loose coupling between interacting services. It emphasizes implementation of components as modular services that can be discovered and used by clients.

The Services generally have the following characteristics [25]:

- Services individually provide a specific functionality and they can be integrated (composed) to provide higher level services promoting re-use of existing functionality.

- Services communicate with their clients by exchanging messages. Service interface defines the messages they can accept and the responses they can give.
- Services may be completely self-contained, or they may depend on the availability of other services, or on the existence of a resource such as a database.
- Services advertise (publish) details such as their capabilities, interfaces, policies, and supported communications protocols. Implementation details such as programming language and hosting platform are of no concern to clients, and are not revealed.

SOA consists of following three components:

- ***Service Publisher*:** Also known as *service provider* or simply *service*. The service publisher is a software component that publishes its services and interface requirements with a service broker
- ***Service Broker*:** Also known as the *registry*. The service broker is responsible for enabling service discovery
- ***Service Subscriber*:** Also known as the *client* or *service consumer*. The service subscriber is a software component that subscribes to a service by discovering the available services that meet some predefined criteria on the network and binding to the service publisher.

Figure 2.2 illustrates the service interaction between the three basic components of SOA: The service registry, service consumer and service provider. The process begins by a service provider advertising its service through a well-known registry (UDDI registry) (Interaction 1 in Figure 2.2). Then the client which may or may not be a service, queries the registry for a service that meets its needs (Interaction 2 in Figure 2.2). The registry, in response, sends a matching set of services back to the client. The client then selects one service provider, sends a request message to it using any mutually recognized protocol (Interaction 3 in Figure 2.2). Finally, the service provider responds with the results of the requested operation or with a fault message.

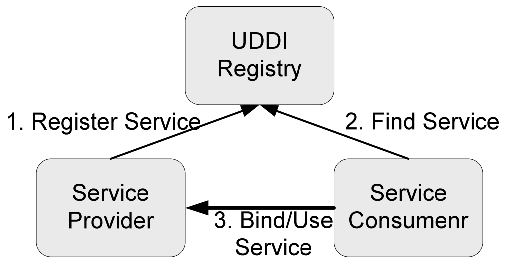


Figure 2:2 Service Oriented Architecture

The publishing of web services are generally done through WSDL (Web Service Description Language). It is an XML-based language that provides a model for describing web services and how to access them. It is the standard for web service description language. WSDL is also used to locate web services. The registry where these WSDL files are published uses UDDI service. UDDI (Universal Description, Discovery and Integration) is a directory service where businesses can register and search for Web services. The communication between registry, client and the service typically use HTTP or SOAP (Simple Object Access Protocol).

2.3.2 Web Services (WSs)

Web Service specifications define an interoperable platform supporting a SOA. WSs are typically APIs accessible via HTTP. They are executed on a remote system that hosts the requested service. According to W3C, a Web service is a software system designed to support interoperable machine-to-machine interaction over a network. The interface of a web service is described using WSDL. Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Web Services vs. Middleware

Web services are a logical evolution of software components and middleware [29]. The problem of heterogeneity and distribution across a network is addressed by middleware community by developing so called middleware services. Middleware services provide a standard programming interface to assure portability and standard protocols to provide interoperability among programs. As the name implies, the middleware is located between the application programs and the network operating system.

One can observe certain similarities between middleware and web services. Like Middleware, web services also provide a standard set of protocols (HTTP, SOAP, WSDL and UDDI) and are accessible through a single unified interface. Also, similar to middleware systems, Web services were introduced with the intention to resolve heterogeneity problems in distributed systems. For example both middleware and web services try to resolve problems such as transformation between different encoding types, or mapping object models and type systems etc.

A major difference of web services, however, is that, unlike middleware systems, they aim at preventing vendor incompatibilities. Therefore two of the significant advantages of web services over middleware is that web services are both vendor and platform independent. Middleware however, is proven to be very successful in tacking important problems in distributed systems such as communication, coordination, control, information management, system management, computation, etc. Web services still have a long way to go to reach the same success. Therefore, the web services also comply with the same definition of middleware, excluding the fact that they are not yet accepted everywhere for critical implementations, reflecting the lack of maturity of the technology.

Web services are very similar to normal software components. Therefore they exhibit characteristics of both software components and middleware. Web services are therefore capable of replacing the distributed middleware services functionalities that typically use a LAN network with functionalities that use the Internet [29]. This vision however, is not yet a reality.

2.3.3 Web Service Composition

Sometimes it becomes necessary to combine functionality of several web services to fulfill the need of a given client or when the implementation of a web service's business login involves the invocation of other web services. Such a service built from multiple web service is called a composite service and the process of developing a composite service is called service composition. The components of a composite service can in turn be an elementary service or a composite service.

Web services Modeling and composition is one of the fundamental ways of offering more advanced, extensible and scalable service provided by a system. In resource constrained environments such as sensor networks or DSC networks however, web service composition is more challenging than that in a resource abundant traditional computer network. There are several detailed surveys on web services and composition [28, 50, 51], and below we discuss in more detail efforts that are more closely related to our work.

The authors of [52, 53] describes composition and filtering of semantic web services using OWL-S [54], the web ontology language for services. Here, the composition is semi-automatic. I.e. set of possible matching services are offered to the user at every step of the composition process and the user manually selects one. However, these approaches are not targeted at energy constrained environments.

Few approaches for service composition have been proposed for sensor networks. Semantic Streams [55] is one of the significant works proposed in this area. Here, authors propose a novel method for service chaining or composition using backward chaining aided by logic programming. This method is used for automated service composition. Authors of [56] propose a service composition method where service compositions that are more persistent in near future are identified to minimize re-composition thereby reducing computation and communication cost. In [57], semi automated service composition is proposed based on dynamic flow control. Filters are used on the wires (logical conditions) to allow the user to manually block data flow to achieve the service desired. Authors of [58] propose an abstract task graph where its abstract tasks and abstract channels are mapped to services and connections respectively, in the service graph. This work is similar to our proposed static process model in the sense, graph construction is not dynamic.

There are two types of service compositions: static composition and dynamic composition.

(i) Static Web Service Composition

In static composition, the requester should build an abstract process model before the composition planning starts. The abstract process model includes a set of tasks and their data dependency. Each task is includes the necessary functionality to search the real atomic web service that fulfils its task. Therefore, the web service selection and binding is automatic even if the process model is static. Static web service composition is less computationally expensive than its dynamic counterpart, as process model is predefined. This composition method is mostly suitable for an environment where only limited set of services are offered and energy and latency are critical parameters.

The need for defining workflows for business logic has lead to development of the business process execution language for web services BPEL4WS [26]. As IBM defines, BPEL4WS provides a language for the formal specification of business processes and business interaction protocols. By doing so, it extends the Web Services interaction model and enables it to support business transactions. BPEL4WS defines an interoperable integration model that should facilitate the expansion of automated process integration in both the intra-corporate and the business-to-business spaces. However standard ways for service discovery like BPEL4WS and DAML-S Service Model [27] are focused on representing service compositions where both flow of a process and bindings between services are known a priori.

(ii) Dynamic Web Service Composition

The Web service composition is a highly complex task, and it is already beyond the human capability to deal with the whole process manually. Therefore, building composite Web services with an automated or semi-automated tool is critical. Figure 2.3 illustrates a typical framework for dynamic web service composition [28]. Service requestors are the consumers of the services provided by service providers. The translator translates between the external languages used by the participants and the internal languages used by the process generator. The process generator attempts to generate a process model or a plan that will fulfill a given request. It is possible that process generator generate multiple plans for a service request. Then, the evaluator evaluates all the plans and selects the best plan for execution. The execution engine finally executes the selected plan and returns the results to the service provider.

In our thesis work, we propose to perform data aggregation in DSC network using web service composition. Being a network for a very specific set of applications, DSC nodes can only offer limited set of web services (Ex: face detection, object tracking etc.). DSC network also pose requirements such as real time guarantees of service delivery and minimum overall energy consumption within network. Therefore DSC networks are perfect candidates to take advantages of static web service composition. In our proposed composition framework, we define our process models to be static but service selection and binding is still dynamic.

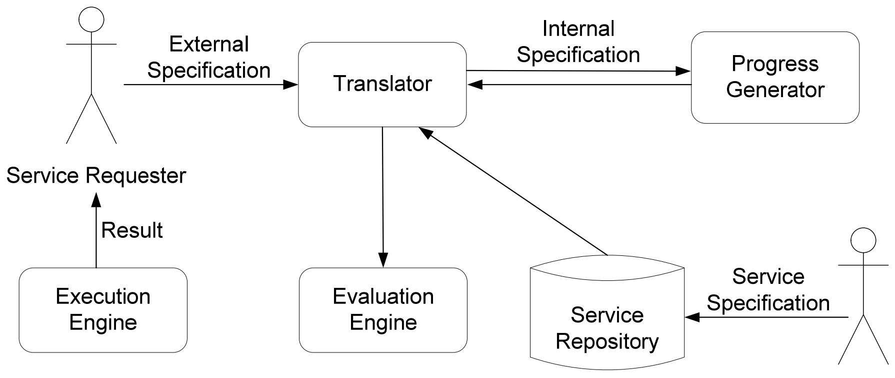


Figure 2:3 Service Composition Framework

Neither web service modeling nor composition has been studied for DSC networks to date. In this thesis work we provide both of these for a DSC network for the first time. In our work, development of dynamic service composition is not necessary due to the fixed and predefined set of services offered by the system. Therefore we prefer a static process model over dynamic composition. However, the concrete services realization and binding is performed at the run time dynamically.

2.4 Query Processing APIs for Distributed Environments

There have been some recent publications in the database and systems communities on query processing in sensor networks [1, 30, 31, 32, 33]. These works highlight the importance of power sensitivity.

Current queries proposed for sensor networks [1, 2, 32, 33, 59, 60] are usually adaptations of SQL queries for traditional computer networks. Due to properties such as limited power, low communication cost, low computation capability and low bandwidth requirements, adapting existing query processing techniques developed for traditional distributed systems adapting these query APIs in resource constrained environments is a tedious task.

The work presented in [59] primarily focuses on when and how often data are physically acquired (sampled) and delivered to query processing operators. A significant reduction of power consumption is achieved by focusing on locations and cost of acquiring data. The authors propose the novel idea of *acquisitional query processing (ACQP)* based on the fact that smart sensor have control over where, when and how often data is sampled. Authors have designed and implemented an ACQP engine called TinyDB [1], which is a distributed query processor that runs on top of every sensor node in the network. We will discuss TinyDB in detail in section 3.5. Authors of [2] extend the functionality of TinyDB by more sophisticated data analysis capability. Three main applications: topographic mapping, wavelet-based compression and vehicle tracking are used to illustrate these sophisticate data analysis tasks. Yong Yao *et. al.* in [32,] present the Cougar approach where a sensor network is tasked through declarative queries. Given a user query, the query optimizer generates an efficient query plan that vastly reduces resource consumption. In [33], authors introduce precision into queries to allow the user full control of the tradeoff between precision and energy usage. By employing the notion of value prediction at the base station, the need for constant communication of sensed values from the sensor devices to the base station is avoided conserving valuable battery lifetime.

In summary, the predominant focus in sensor networks to date has been on power aware in-network query processing, particularly selection and aggregation. In our thesis work we too endorse power aware in-network processing. To our knowledge, no prior work addresses this issue for DSC networks.

Query processing APIs for sensor networks (ex: TinyDB[1] query processing API) typically provide an SQL query interface that incorporates the concepts of sampling intervals, monitoring periods into SELECT-FROM-WHERE clause etc. One of the main SQL query processing function provided is data aggregation. A sample query run on a typical sensor network would have a format similar to following [33]:

```
SELECT AggregationFunction
FROM Sensordata s
WHERE s.loc in R
DURATION D
EVERY t
```

Where *Aggregation Function* can be aggregates such as AVG, SUM, MAX, and MIN. *s* specifies the sensor types, *R* is the query region, *D* gives the query runtime and *t* specifies the sampling rate.

A sample query that calculates average temperature in Room1 that is run for 30 seconds every second would look like the following:

```
SELECT AVG(s:temperature)
FROM s.temperature
WHERE s.loc in Room1
DURATION 30s
EVERY 1s
```

The work presented in [59] primarily focuses on when and how often data are physically acquired (sampled) and delivered to query processing operators. A significant reduction of power consumption is achieved by focusing on locations and cost of acquiring data. The authors propose the novel idea of *acquisitional query processing (ACQP)* based on the fact that smart sensor have control over where, when and how often data is sampled. Authors have designed and implemented an ACQP engine called TinyDB [1], which is a distributed query processor that runs on top of every sensor node in the network. We will discuss TinyDB in detail in section 3.5. Authors of [2] extend the functionality of TinyDB by more sophisticated data analysis capability. Three main applications: topographic mapping, wavelet-based compression and vehicle tracking are used to illustrate these sophisticate data analysis tasks. Yong Yao *et. al.*

in [32,] present the Cougar approach where a sensor network is tasked through declarative queries. Given a user query, the query optimizer generates an efficient query plan that vastly reduces resource consumption. In [33], authors introduce precision into queries to allow the user full control of the tradeoff between precision and energy usage. By employing the notion of value prediction at the base station, the need for constant communication of sensed values from the sensor devices to the base station is avoided conserving valuable battery lifetime.

In summary, the predominant focus in sensor networks to date has been on power aware in-network query processing, particularly selection and aggregation. In our thesis work we too endorse power aware in-network processing. To our knowledge, no prior work addresses this issue for DSC networks.

One of the advantage of DSC network and Sensor networks query APIs over traditional distributed networks counterpart is that the query API for sensor (and DSC) networks needs to handle limited set of queries as such networks are application specific. To our knowledge, no prior work for DSC networks addresses these issues. Many of the research directions in DSC networks to data have converged toward object detection, motion tracking and alike.

Providing a SQL like query interface for DSC networks is a more challenging task than providing the same for a traditional sensor network. This is because the data that needs to be aggregated are not simple data like temperature or humidity. Rather they are more complex images and videos. We need to provide separate data aggregation methods and query APIs specific to DSC networks.

2.4.1 TinyDB

In this section, we discuss TinyDB [1], which is an acquisitional query processing system for sensor networks. TinyDB runs on the Berkeley mote platform, on top of the TinyOS [71] operating system. TinyDB includes many features of a traditional query processor such as ability to select, join, project, and aggregate data. Acquisitional techniques are used to minimize power consumption in the network.

In TinyDB, sensor tuples belong to table *sensors*. Logically, table contains one row per node per instant in time, with one column per attribute (eg. light, temperature, etc.) that the device can produce. These records are acquired only as needed to satisfy the query and only stored for a short period of time. Physically, the sensor table is distributed across the sensor nodes in the network. Each device stores its own readings in the partition of sensor table it contains.

Basic query language features of TinyDB include SQL queries consist of SELECT-FROM-WHERE-GROUPBY clause that support selection, join, projection and aggregation. The FROM clause may refer to both the *sensors* table as well as stored tables, which we call materialization points. Queries also support *sample interval* specification where tuples need to be generated. The *epoch*, or the time between start time of each sample period allows minimum power consumption.

A sample query in TinyDB looks like as follows:

```
SELECT nodeId, light, temp
FROM sensors
SAMPLE PERIOD 1s FOR 10s
```

This query specifies that each device should report its own id, light, and temperature readings (contained in the virtual table sensors) once per second for 10 seconds. Nodes initiate data collection at the beginning of each epoch, as specified in the SAMPLE PERIOD clause. Results of the query stream to the root (sink) of the network via multihop topology. The output essentially is a stream of tuples clustered into 1s intervals. Nodes in TinyDB run a simple time synchronization protocol to agree on a global time base that allows them to start and end each epoch at the same time.

TinyDB also supports aggregate query formulation. A sample query that finds the average volume of a room where this average volume exceeds a given threshold every 30s is given as follows:

```
SELECT AVG(volume),room FROM sensors
WHERE floor = 6
GROUP BY room
HAVING AVG(volume) > threshold
SAMPLE PERIOD 30s
```

In addition to aggregates over values produced during the same sample interval (eg: COUNT, AVG, MAX, MIN etc.), *temporal aggregates* are also supported by TinyDB. For example, the following query measures the maximum volume of a given location over a given time (30s in this case) once every 5 seconds.

```
SELECT WINAVG(volume, 30s, 5s)
FROM sensors
SAMPLE PERIOD 1s
```

This is an example of a sliding-window query common in many streaming systems.

In our work we too propose aggregate queries including temporal aggregates. We also provides language primitive that allows for sophisticated query processing such as detecting number of faces of a given image.

2.4.2 MORE

In MORE [72], authors are proposing a service orchestration (composition) mechanism applied to services on top of a DPWS-based middleware. The Device Profile for Web Services (DPWS) implements a subset of WS-* specifications in order to make the advantages of the Web Service architecture available to a growing embedded systems market. The approach is complementary to the rather complex and resource intensive Web Service Business Process Execution Language (WS-BPEL) and focuses on service orchestration on resource constrained devices deployed in hierarchical network topologies. It follows the paradigm of Service Oriented Architectures by adopting DPWS and OSGi for the middleware design.

MORE middleware currently supports a variety of functionalities such as WS addressing, WS discovery, WS metadata exchange. The Core is a major component in MORE. It provides access to the metadata and WSDL definitions of other services hosted at remote devices. The events triggered by these remote services use the publish/subscribe mechanism provided by the eventing services. The execution services offer means to invoke operations on other services and in turn to receive invocations from requesters. The primary network interface implements the standard SOAP over HTTP binding for message exchange and is utilized for web service invocation.

The disadvantage of MORE however is that it provides more generic services, compared to application specific services desired in a DSC network. Due to this, it is impractical to provide a SQL query API for MORE middleware.

In our thesis work, we go one step further to MORE by proposing a SQL API for web service composition enabled environment and by using Distributed OSGi. Users will have the opportunity to submit queries to the network, where they are converted to web service compositions transparently to the user to provide the results.

2.5 OSGi and Distibuted OSGi

2.5.1 OSGi (Open Source Gateway initiative)

OSGi [34] is a dynamic module system for java. It provides component base environment and offers component life cycle such as installing, starting, stopping, and uninstalling. Applications or components (represented as of bundles for deployment) can be installed, started, stopped, updated and uninstalled without requiring a reboot. This component life cycle provides dynamic functional objects that can be used, obtained, removed, refreshed and replace dynamically. OSGi is a java platform mostly being used in embedded systems.

The two main component of the OSGi are the components that are represent as bundles and the OSGi framework. OSGi bundles are behavioral components that offer libraries statically and services dynamically. OSGi container deploys bundles in jar format. The jar file has two basic parts, code and the manifest file. The classes in the jar file loaded into OSGi framework using java class loaders. All the bundles deployed in an OSGi container run in one Java Virtual Machine (JVM).

OSGi also has a service registry. Using this service registry, bundles can publish and/or consume services. This service registry enables the service oriented architecture on top of JVM called "SOA in a JVM". However unlike SOA which rely on web services for communication, OSGi services are published and consumed within the same java virtual machine.

This framework illustrated in Figure 2.4 is conceptually divided into the following areas:

(i) *Bundles:* Bundles are normal jar components with extra manifest headers.

(ii) *Services:* The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects (POJO).

(iii) *Services Registry:* The API for management services (ServiceRegistration, ServiceTracker and ServiceReference).

(iv) *Life-Cycle:* The API for life cycle management for (install, start, stop, update, and uninstall) bundles.

(v) *Modules:* The layer that defines encapsulation and declaration of dependencies (how a bundle can import and export code).

(vi) *Security:* The layer that handles the security aspects by limiting bundle functionality to pre-defined capabilities.

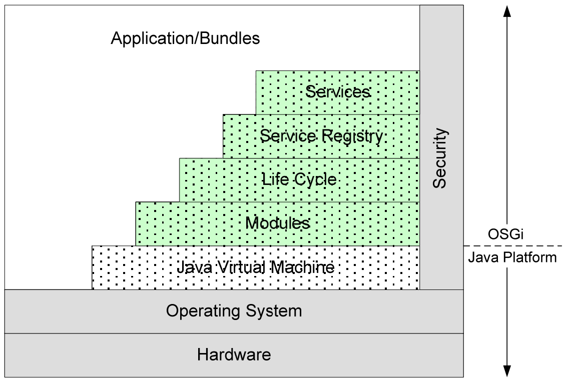


Figure 2:4 OSGi Framework

Many popular technologies including IBM Websphere, IBM RSA, BEA, Eclipse, Apache (struts 2), Spring (Spring OSGi), Simens, Nokia, BMW, Cisco and many more use OSGi in their technologies.

2.5.2 Distributed OSGi

If a client of a service resides outside this OSGi container then the service bundle need to have distributed capability. Apache cxf-dosgi [37] is a new services framework that enables distributed capability for OSGi bundles. In our thesis work we use this frame work to develop and deploy web services as OSGi bundles. Apache CXF also provide front end programming APIs for various protocols.

2.6 Face Detection

Approaches for face detection proposed by the research community mostly deal with faces at arbitrary scale, though most assume upright faces. Of these methods, Schneiderman and Kanade [61] use statistical methods to detect faces in a 3-D setting. Their method consider only three face orientations and each orientation is treated as a different object. Rowel and Kanade in [62], uses neural network based filters for face detection. In [63], Papageorgiou *et a*l propose a general object detection scheme that uses statistical learning and a wavelet representation. Most of these traditional distributed are deeply connected to their design constraints and reengineering them to a power sensitive network often requires a great deal of effort.

With rapid advances in hardware miniaturization Wireless Multimedia Sensor networks (WMSN) and DSC networks also provide face detection capabilities that aid in a variety of applications including multimedia surveillance, traffic monitoring and environmental monitoring. T. Yan et al in [65] present a distributed search system over a camera sensor network. In their implementation, iMote2 sensors are used for nodes to sense, store and search for information. In [66], authors propose a network of dual-camera nodes that is used for retrieving misplaced objects in a home environment. Both low power and high power camera nodes are used in the platform. A low cost, open source computer vision platform called CMUCam3 [69] is presented in [67]. A study for an embedded implementation of boosting of boosting based face detection into hardware is described in [68]. However, one of the most significant and most current works of this area is that presented by Viola and Jones [36] for real time face detection that is capable of achieving high detection rates. A light weight version of the algorithm is implemented in CMUCam3 [70]. This algorithm is essentially a feature based approach, where a classifier is trained for Haar-like rectangular features. Here, the images are scanned at different scales and positions. The rapid detection rate of this algorithm is achieved by a novel technique called *integral image* formation. Also, a series of classifiers are organized in a cascade from simple to more complex classifiers towards the end for feature recognition. A region to be declared to be containing faces should pass all the classifiers in the cascade. This way, easier regions are eliminated early in the cascade from further processing, if classified for not containing a face while the difficult regions are operated on by more complex classifiers. This greatly speeds up the detection process without compromising on the accuracy and provides high detection rate. This algorithm provides performance comparable to the existing best face detector systems such as [61, 62, 63] but with orders of magnitudes faster than any of these systems. On a conventional desktop, it can detect faces at 15 frames per second. In our thesis work we use this algorithm to be implemented in individual camera nodes.

3 SYSTEM ARCHITECTURE AND ALGORITHMS

This chapter provides the detailed description of our contributions made to the thesis: the proposed web service modeling and composition in DSC networks, the complete DSC middleware system architecture, Dynamic loading and unloading services, and SQL Language primitives for DSC networks. In the rest of the chapter we may use terms *smart camera network* and *DSC network* interchangeably to refer to the distributed smart camera network.

3.1 Web Service Enabled Smart Camera Network Architecture

3.1.1 Smart Camera Network Architecture

The DSC network that we consider in our implementation is of hierarchical network architecture. As proven by previous research, hierarchical architecture is an excellent choice for data aggregation networks. Compared to flat networks, where readings are sent directly to a single sink, hierarchical network allows employing tree based aggregation strategies reducing communication and computation cost considerably. It also allows for better load distribution among participating nodes for a given query. A hierarchical network is easily scalable to any depth.

Figure 3.1 depicts the network architecture of our DSC network. As illustrated in the figure, in our hierarchical architecture, a high-end server is directly connected to the root camera node of the hierarchy. This high-end server is also connected to the World Wide Web (WWW). Users issue queries through WWW to this server. All the smart camera nodes reside at different layers of the hierarchical network. It is therefore the responsibility of the server to accept queries from the users, communicate with camera nodes through the root camera node to provide the results back to the users, acting as the interface to the DSC network for users.

In our proposed hierarchical DSC network, the camera nodes with higher number of children are more powerful in terms of processing capability, memory capacity etc. than the nodes with less number of children. The reason for this is that camera nodes with more child camera nodes in the hierarchy need to have the capability to aggregate large amounts of data from multiple sources. Intuitively, the least powerful nodes are at the leaf level of the hierarchy.

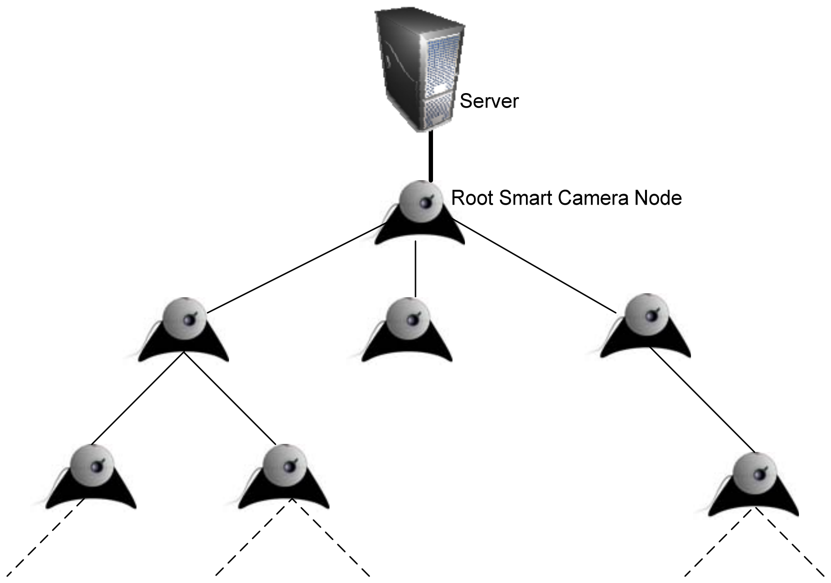


Figure 3:1 Hierarchical DSC Network Architecture

3.1.2 Layered Middleware Architecture of a Smart Camera Node

A layered architecture is often used in order to support software flexibility in different levels. Figure 3.2 (b) illustrates our proposed layered middleware layered architecture of a smart camera node which we adapted from the general purpose middleware architecture proposed by Schmidt et al. in [73] (Figure 3.2 (a)).

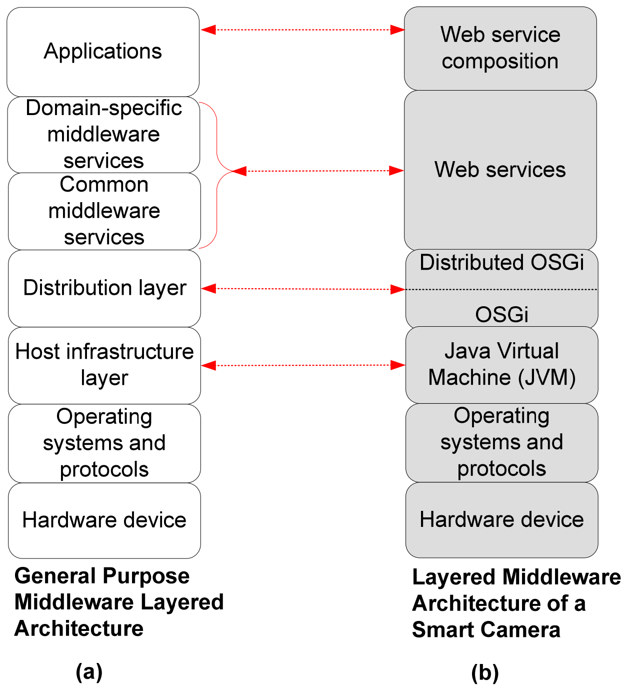

Applications
Domain-specific middleware services
Common middleware services
Distribution layer
Host infrastructure layer
Operating systems and protocols
Hardware device
General Purpose Middleware Layered Architecture
(a)
Web service composition
Web services
Distributed OSGi
OSGi
Java Virtual Machine (JVM)
Operating systems and protocols
Hardware device
Layered Middleware Architecture of a Smart Camera
(b)

Figure 3:2 Layered Middleware Architecture of a Smart Camera Node

The layers of our layered middleware architecture for a camera node are as follows:

(i) Hardware Device Layer

The hardware device layer of a smart camera node contains the sensing module, processing unit and the communication unit as given in Figure 2.1 in chapter 2. The hardware device provides sufficient processing power and fast memory for processing the images in real time while keeping power consumption low.

(ii) Operating System and Protocols Layer

This layer provides drivers for accessing the device as well as concurrency, process and thread management and inter-process communication.

(iii) Java Virtual Machine (JVM) Layer

This layer provides the portability and platform independence to the middleware framework. It hides the lower level system calls and supports communication and concurrency mechanism. It also provides the object oriented capability to the higher layers. This is the analog to the host infrastructure layer of the general middleware layers depicted in Figure 3.2 (a).

(iv) OSGi Layer

This layer brings modularity to the java platform which enables creation of highly cohesive and loosely coupled modules that can be individually developed, tested, deployed, updated and managed with almost no impact on other modules in the layer they reside. It also minimizes the complexity of the platform independence by using the java programming language, one of the most popular programming languages today. One of the promising factors that lead us to the selection of OSGi in our framework is that it is a widely used and commercially available framework for embedded systems programming. OSGi is used for providing lower level of service compositions at the device level. More detailed description of the OSGi layer can be found in section 3.1.3.

(v) Distributed OSGi (D-OSGi) Layer

D-OSGi layer is used for providing distribution capability to the underlying layers. This is discussed in detail under section 3.1.3.

(vi) Web Service Layer

This layer allows for exposing services offered by D-OSGi as web services. The main objective of implementing a web service layer in our layered node architecture is to provide more scalability, availability, platform and language independency to the system. Camera nodes in the Smart Camera network communicate with each other through these web service calls.

(vii) Web Service Composition Layer

The elementary services provided by smart camera nodes can be chained into more complex web compositions to provide more sophisticated functionality. Web service compositions in our system mostly come in the forms of complex data aggregations. User can invoke both composed services and elementary services to request a domain-specific functionality. Web service composition layer performs web service composition services needed for these data aggregations.

Our selection of the proposed layers and technologies are greatly affected by the recent work done related to our work, both in research community and in industry. Researchers often have emphasized one key difference between middleware developed for resource constrained environments and general purpose middleware: middleware for resource constrained environments focus on reliable services for ad-hoc networks and energy awareness [4]. According to a survey conducted by Molla and Ahmed in [75], most implementations of middleware for resource limited wireless sensor networks are based on TinyOS [71, 76], a component-oriented, event-driven operating system for sensor nodes. Many interesting middleware approaches have been implemented and evaluated for sensor networks. The spectrum ranges from a virtual machine on top of TinyOS, hiding platform and operating system details to more data centric approaches for data aggregation and data query.

Two of the works that is more aligned with our work are Cougar [32] and TinyDB [1] which follow the data-centric approach, integrating all sensor network nodes into a virtual database system where the data is stored distributively among several nodes. However, web service modeling and composition on top of middleware is one of our novel contributions that do not exist in related work. Use of a web service layer solves the critical problems of scalability, interoperability, and node heterogeneity. We also selected third-party components, OSGi and D-OSGi, in our software framework as they are well-developed commercially available frameworks for embedded systems that allow us to expose functionality as services in a resource constrained distributed environment.

3.1.3 Third Party Components

(i) OSGi

OSGi [34] is a component based framework specification that brings modularity to the Java platform. An OSGi container enables the creation of highly cohesive, loosely coupled modules that can be composed into larger applications. Figure 2.4 in Chapter 2 illustrates how OSGi is built on top of the Java Virtual Machine (JVM) with modules definition. OSGi framework provides a lifecycle of modules, a service registry, security, and a set of services for building modular applications. At the lowest level deployment unit of the OSGi container is called *bundle* which is consist of basic code and the manifest file. Manifest file contains the OSGi specific metadata such as unique name, version, dependencies and other deployment details. OSGi bundles are deployed as common JAR file format. OSGi life cycle layer allows these bundles to be installed, started, stopped, and uninstalled from the device.

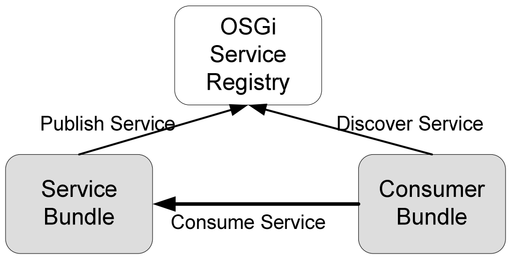


Figure 3:3 SOA in JVM

As shown in Figure 3.3, OSGi service registry allows service producer bundles to publish services and consumer bundles to consume those published services enabling Service Oriented Architecture (SOA) in OSGi framework. However, unlike the many interpretations of SOA, which rely on web services for communication, OSGi services are published and consumed within the same JVM. Thus, OSGi is sometimes referred to as 'SOA in a JVM'.

(ii) Distributed OSGi

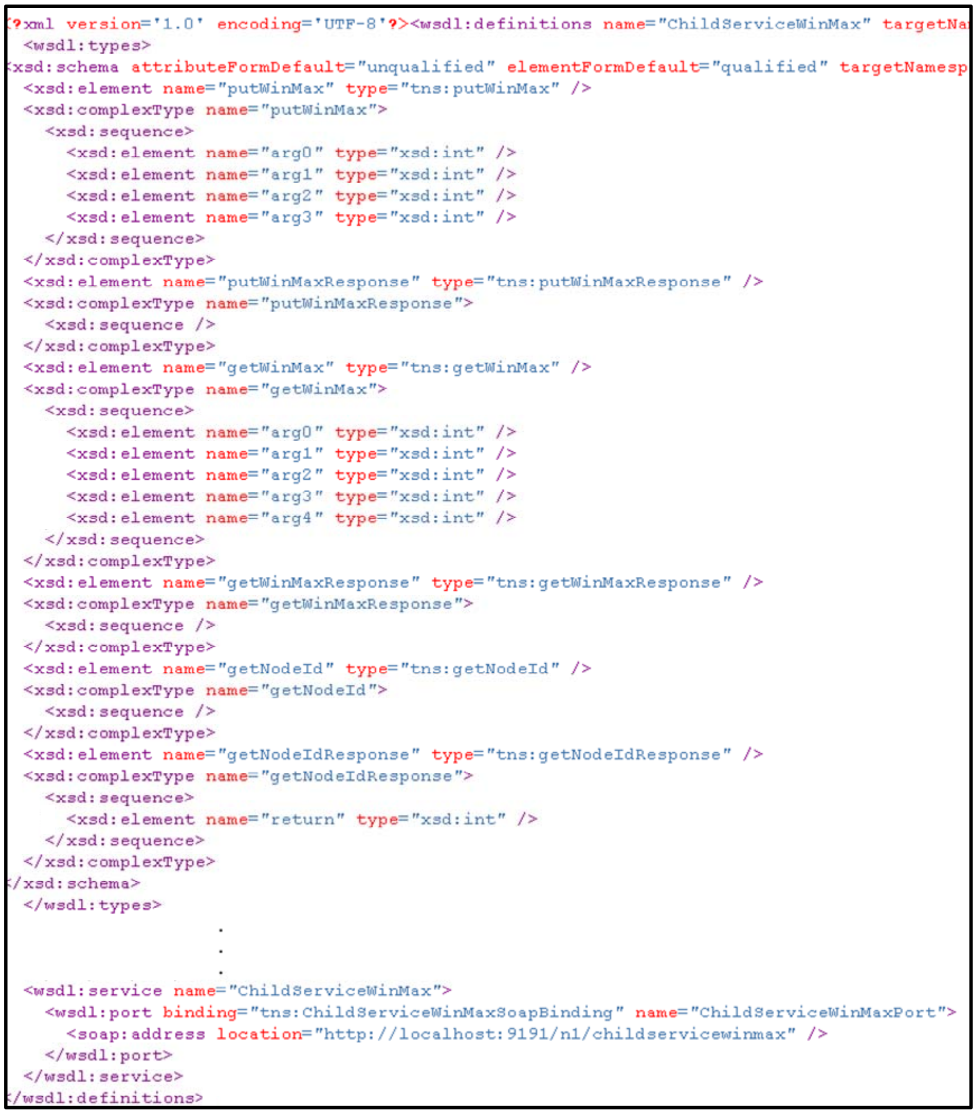
If a client of a service resides outside this OSGi container then the service bundle need to have distributed capability. Apache cxf-dosgi [37] is a new services framework that enables distributed capability for OS-Gi bundles. This allows servies developed as OSGi bundles to be published as web services.

3.1.4 Smart Camera Node Services

A smart camera node provides two types of services*: local services* and *web services*. Local services can only be used by the node offering the given local service. They are not offered for other outside camera nodes. Web services on the other hand, are services offered by a given camera node that are accessible by outside camera node. Both local and web services offer their own APIs.

(i) Web Services

Each camera node offers a variety of web services and each service contains multiple methods that allow accepting requests from users (or smart camera nodes) and sending requested data back to the user. Generally a web service has three operations: one to get the device id, one to get camera readings and another to put (send) readings. A web service may provide a basic query functionality, aggregate query functionality or a core service. Each web service has a unique endpoint and is associated with its own web service description language (wsdl) file that defines the interface details for accessing that service. The D-OSGi layer generates these wsdl files from the APIs of the services. Figure 3.4 shows the wsdl file generated for the WinMax web service.



```
<?xml version='1.0' encoding='UTF-8'?><wsdl:definitions name="ChildServiceWinMax" targetNa
  <wsdl:types>
<xsd:schema attributeFormDefault="unqualified" elementFormDefault="qualified" targetNamesp
  <xsd:element name="putWinMax" type="tns:putWinMax" />
  <xsd:complexType name="putWinMax">
    <xsd:sequence>
      <xsd:element name="arg0" type="xsd:int" />
      <xsd:element name="arg1" type="xsd:int" />
      <xsd:element name="arg2" type="xsd:int" />
      <xsd:element name="arg3" type="xsd:int" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="putWinMaxResponse" type="tns:putWinMaxResponse" />
  <xsd:complexType name="putWinMaxResponse">
    <xsd:sequence />
  </xsd:complexType>
  <xsd:element name="getWinMax" type="tns:getWinMax" />
  <xsd:complexType name="getWinMax">
    <xsd:sequence>
      <xsd:element name="arg0" type="xsd:int" />
      <xsd:element name="arg1" type="xsd:int" />
      <xsd:element name="arg2" type="xsd:int" />
      <xsd:element name="arg3" type="xsd:int" />
      <xsd:element name="arg4" type="xsd:int" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="getWinMaxResponse" type="tns:getWinMaxResponse" />
  <xsd:complexType name="getWinMaxResponse">
    <xsd:sequence />
  </xsd:complexType>
  <xsd:element name="getNodeId" type="tns:getNodeId" />
  <xsd:complexType name="getNodeId">
    <xsd:sequence />
  </xsd:complexType>
  <xsd:element name="getNodeIdResponse" type="tns:getNodeIdResponse" />
  <xsd:complexType name="getNodeIdResponse">
    <xsd:sequence>
      <xsd:element name="return" type="xsd:int" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
  </wsdl:types>
                    .
                    .
                    .
  <wsdl:service name="ChildServiceWinMax">
    <wsdl:port binding="tns:ChildServiceWinMaxSoapBinding" name="ChildServiceWinMaxPort">
      <soap:address location="http://localhost:9191/n1/childservicewinmax" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Figure 3:4 WSDL File for WinMax Web Service

Following are the web services a camera node provides:

a. Raw Data Collection Web Services

RawDataService collects raw image and video streams and send back to the user the number of faces detected in those images. Within this service, three methods are offered: *getNodeId()* to get the node id of the camera sending raw image data(readings) , *getRawData()* to accept image data request from another node and *putRawData()* to accept requested image data sent from another node.

b. Image Data Aggregation Web Services

These services provide various aggregation services on raw data. Each camera node offers four aggregation services: *AverageService*, *MinService*, *MaxService*, *WinMaxService, WinMinService, WinAverageService* for calculating average, minimum, maximum temporal maximum, temporal minimum and temporal average of the number of faces detected of raw image data respectively. Intuitively, all these aggregation services in turn call face detection service to find number of faces of the given image forming a web service composition.

c. Dynamic Loading and Unloading Web Services

Two service loading and unloading services, *LoadService* and *UnloadService* are offered by a given camera node. These services allow starting services when a query is received and stopping services when query is serviced respectively, saving valuable energy and memory spent on services continuously running during network lifetime. *LoadService* offers the method *invokeLoadService(ServiceAPI, ServiceImpl)* that will start the requested service given by service API *ServiceAPI* and service implementation *ServiceImpl*. Sections 3.2.2 and 3.2.3 describe dynamic loading and unloading in more detail.

(ii) Local Services

Each camera node contains several services that run locally. Since these services are not offered to outside camera nodes in the network, they are not associated with a wsdl file. Following are the local services offered by a camera node:

a. Face Detection Service

Each camera node offers *DetectFaces* service that allows the node to capture image and detect number of faces in them using *getNumFaces()* method. We use Viola Jones face detection algorithm [36] to run on camera nodes as it is so far the best algorithm to achieve high detection rates for real time face detection. A lightweight version of the algorithm is available for CMUCam3[70] cameras.

b. Child Communication Service

Each camera node can act as a child when communicating with its parent node in the hierarchy. The service *ChildService* offers *callParentService()* method to achieve this purpose.

c. Parent Communication Service

Each camera node also can act as a parent and call its children nodes using the *callChildServices()* of service *ParentService*.

d. Dynamic Loading and Unloading Services

This is used for dynamic loading and unloading web services. The *LoadService* and the *UnloadServices* are local services and are used to load and unload local bundles dynamically. These bundles can be the bundles offering web services.

Figure 3.5 depicts a detailed view of the proposed single camera node architecture. It shows local services (white rectangles), web services (shaded rectangles), and their corresponding APIs (rounded rectangles). As illustrated in the figure, a camera node is implemented as an *OSGi container*. Services offered by a camera node and their APIs are implemented as *OSGi bundles*. A bundle is a deployable module in OSGi container. A bundle may consist of one or more *packages*. These are simply java packages that contain highly cohesive set of classes. A service implementation and its service API is usually included in separate packages.

In a smart camera node, there are two major methods of representing services, each with its own advantages. First method is to contain both service implementation and service API in one single bundle. The services *ChildClient* and *ParentClient* are two such service bundles depicted in Figure 3.5. This method is suitable when a service has only one implementation. However, since both service API and implementation are defined in the same bundle, both API and implementations need to be deployed (i.e. run), for other services and users to access this service. Second method is to separate the service implementation and API by defining them as separate bundles. For example FaceDetection service and all web services (e.g. *MaxService*, *MinService* etc.) use this implementation method. There are several advantages of this method. First, unlike first method, by separating service implementation from its API, only API bundle needs to be installed for another bundle to start the service or user to access the service, while not starting service implementations. Therefore service implementations can be dynamically loaded and unloaded as needed. Second, this allows for easy development of multiple implementations for the same service API by allowing the application developer to develop separate service bundles for each implementation. Third, since all camera nodes offer same set of services, when a local service of a camera node needs to use a remote service (i.e. a web service of another camera node) it needs to import remote service API. Instead, it simply can import the matching local service API locally.

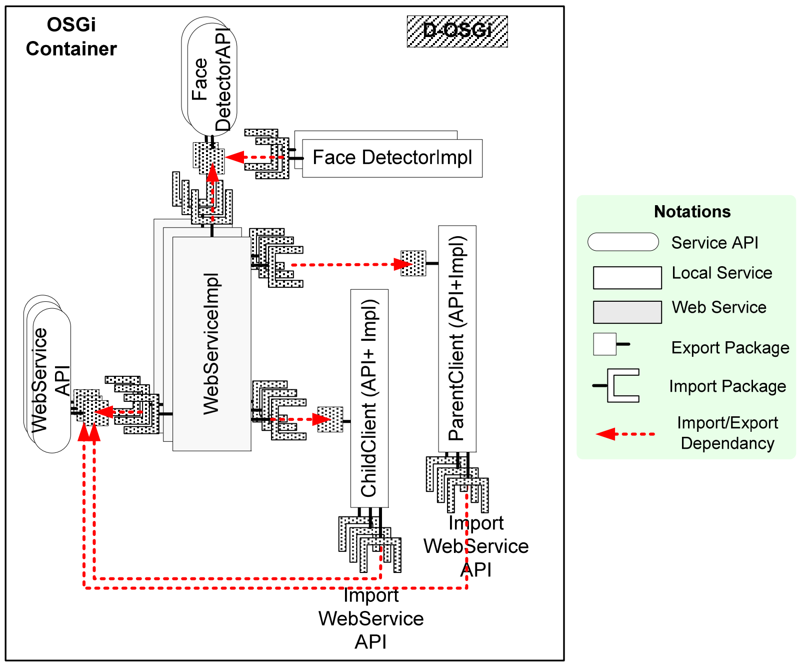


Figure 3:5 Smart Camera Architecture

Services residing in a camera node can make its web services available to other services as well as use remote web services of other camera nodes. To make its service available to outsiders, a service bundle should explicitly *export* its package containing the service API of the implemented service (depicted a square in Figure 3.5). Similarly, for a service to use another service, it must *import* the package that contains the needed service API (depicted as a handle in Figure 3.5). This import/export mechanism models the dynamic plug-in mechanism of services. These import/export dependencies are also depicted in Figure 3.5 as dotted arrows. For example, the service implementation *WebServiceImpl* (e.g. *MaxServiceImpl*) is made available to other services using *WebServiceAPI* (e.g. *MaxServiceAPI*) export mechanism. *WebServiceImpl* imports three service APIs: *ChildClientAPI* for communication with parent node, ParentClientAPI for communication with children nodes and *FaceDetectorAPI* for detecting faces in images.

3.2 Query Model

We choose SQL-style query language as our high level language primitive for the DSC network. The basic SQL query language primitives such as SELECT clause are adapted from TinyDB [1] to suit an image processing environment. The queries are executed on a single table called *Cameras*. This table is horizontally partitioned among camera nodes meaning that each camera node maintains set of data tuples of the table. The table fields are constructed dynamically based on the query. Table data tuples are also inserted dynamically based on camera node readings.

3.2.1 Basic SQL Models Supported by DSC SQL API

The query interface supports three types of quires: the basic queries, the aggregate queries and the temporal aggregate queries. These queries are internally converted to web service compositions to carry out the requested task. This will be discussed in detail under section 3.3.

(i) Basic Select Queries

In general, basic language queries have following format.

```
SELECT {attributes}
FORM Cameras
SAMPLE PERIOUD i FOR j
```

The SELECT clause can specify the attributes that is required as output to the user. Upon receipt of the query, a camera initiate the data collection interleaving the sample period i, FOR the duration j. For example, when the following query is issued, a camera node starts collecting image data every 5 seconds up to 20 seconds.

```
SELECT nodeId,numFaces,epoch
FROM Cameras
SAMPLE PERIOD 5s FOR 20s
```

At each 5th second, camera node will detect an image, call the *DetectFaces* service to detect faces of that collected image and send its node Id, detected faces and the epoch (which 5th second) tuple to the parent node(query sender) as soon as faces are detected. Thus, this node ends up sending five result tuples computed for the data collected at 0th, 10th, 15th, and 20th seconds. In addition to this, upon receipt of the query, a camera node also forwards the query to all its children and as a result, it will also send result tuples received from children nodes from time to time to its parent node. Thus the user will receive a table that contains multiple tuples, five tuples from each camera node.

(i) Aggregate Queries

The aggregate query has the similar format to that of a basic select query and specify an aggregate function AGG, over a single attribute as follows.

```
SELECT AGG(attribute)
FROM Cameras
SAMPLE PERIOD i FOR j
```

For example, the following query will result in a camera node collecting images and detecting faces of the colleted image at every 5 seconds and sending the tuple which contain the maximum number of detected faces.

```
SELECT MAX(nodeId,numFaces,epoch)
FROM Cameras
SAMPLE PERIOD 5s FOR 20s
```

Similar to the basic SELECT, five readings are taken at 0th, 5th, 10th, 15th and 20th second, but the result sent to the parent by each camera node is taken from one of these readings, the tuple with the maximum number of detected faces. The result tuple is sent to the patent of camera node at the 20th second. Similar to the previous case, a camera node will forward query to its children upon receipt of the query, and send tuples received from children nodes back to its parent node upon their receipt. This results in each participating camera node sending one result tuple back to the user.

In addition to MAX aggregate, we also offer MIN, and AVERAGE aggregate clauses for image data aggregation.

(i) Temporal Aggregate Queries

Implementing temporal aggregates more challenging than the previous two because a sliding window for history data needs to be maintained. The temporal aggregate we developed has the following format:

```
SELECT WINAGG(attr)
FROM Cameras
SAMPLE PERIOUD i FOR j WIN SIZE k RES l
```

This perform the AGG function over last k time unites (window size WIN SIZE) once every l time unites, sample interleaving i for duration j. This query gives a running aggregate.

For example, following query will result in the camera node collecting data every 1 second for 10 seconds, while calculating the maximum number of faces detected for images detected in last 5 seconds, at every 2 seconds and sending result tuple back to parent node.

```
SELECT WINMAX(nodeId,numFaces,step)
FROM Cameras
SAMPLE PERIOD 1s FOR 10s WIN SIZE 5s  RES 2s
```

3.2.2 Query Execution Model

As we mentioned earlier, user queries are converted to a web service composition. That is, to run a particular query all web services do not need to be up and running in camera nodes. This is an important factor for a resource constrained environment. Therefore we develop an energy efficient query execution model where queries can be executed with less low energy and memory consumption.

Figure 3.6 show the proposed query execution model applied to a single camera node receiving a query. Upon query receipt, query execution component of the camera uses a static process model to identify the required services and the service composition to run the query. A static process model is highly suitable for this kind of application specific environment which offers fixed set of application specific queries. Once the services are identified, the camera node invokes local service *LoadService* to deploy the required service and invoke each child camera node's remote web service *LoadService* to triggering them to deploy. Then the camera node execute query as a service composition which is propagated down the network. Once the results are collected locally and from children they are sent to the root and *UnloadService* is called to unload the service. Therefore only the relevant services run during query execution and at other times only core services run. This saves valuable energy by eliminating the need of all services running all the time maximizing the network lifetime.

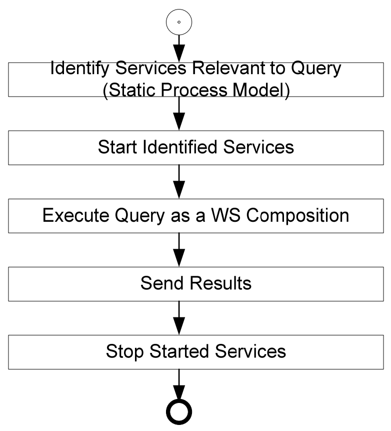

Identify Services Relevant to Query (Static Process Model)
Start Identified Services
Execute Query as a WS Composition
Send Results
Stop Started Services

Figure 3:6 Query Execution Model

3.2.3 Dynamic Loading and Unloading of Services

As mentioned earlier, in our query model, the services required for execution of the query needs to be loaded before a query is executed. When a user issues a query at the server, server identifies the set of services that needs to be running in every node. Then the server invokes the *LoadService* service of the root camera node. As depicted in Figure 3.7, each camera node will do the following things upon invoke of its service *LoadService*:

(i) Invoke local *LoadService* to load the requested service locally.

(ii) Invoke remote *LoadService* on each child camera node, to request to load requested service in the child camera nodes.

(iii) Send *finishedLoaDService* confirmation to parent camera node when it receives confirmation from its local load service and all children camera node load services.

Intuitively, a camera node sends the confirmation of successful load back to its parent only when it and its children camera nodes confirm the loading of requested service. Therefore, when the root camera node receives the confirmation, the whole sub-tree root at it has started the requested services and thus, the root can start propagating user query to camera nodes for execution.

Service unloading executes in a similar fashion. When the server finishes receiving all the results, it will invoke *UnloadService* of the root camera node, which will unload the requested services locally and propagate unload request to its sub-tree.

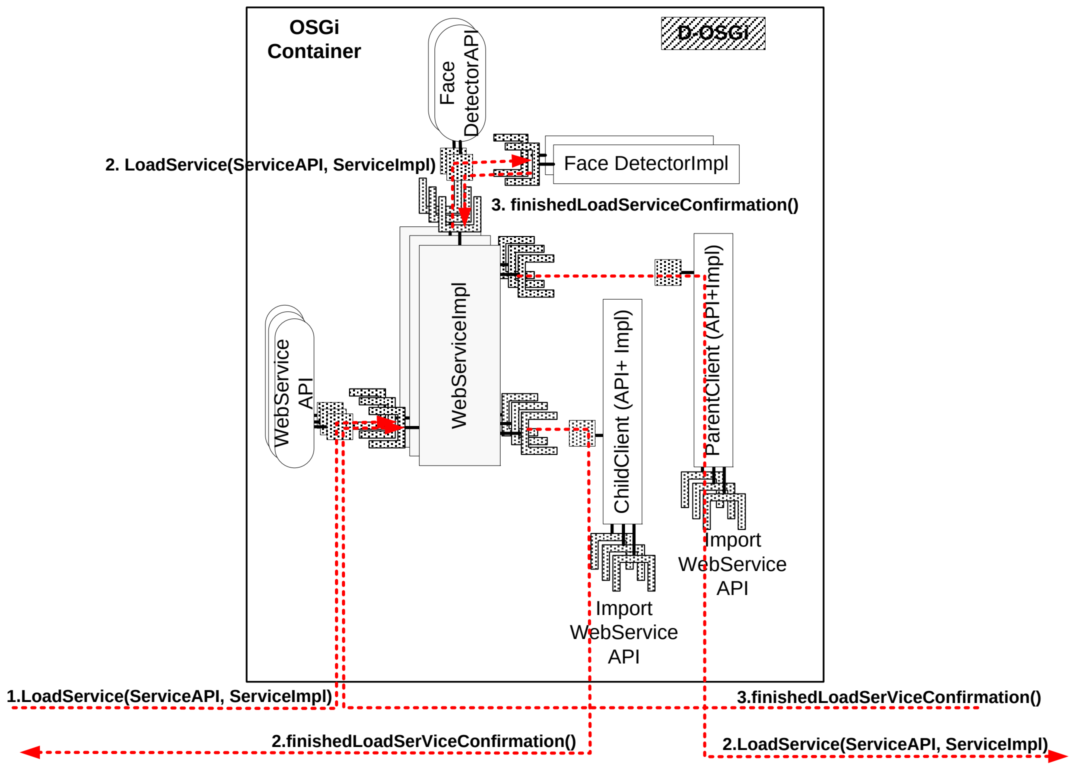

OSGi Container
D-OSGi
Face DetectorAPI
Face DetectorImpl
2. LoadService(ServiceAPI, ServiceImpl)
3. finishedLoadServiceConfirmation()
WebServiceImpl
WebService API
ChildClient (API+ Impl)
ParentClient (API+Impl)
Import WebService API
Import WebService API
1.LoadService(ServiceAPI, ServiceImpl)
3.finishedLoadSerViceConfirmation()
2.finishedLoadSerViceConfirmation()
2.LoadService(ServiceAPI, ServiceImpl)

Figure 3:7 Dynamic Service Loading

3.3 Aggregation Using Web Service Composition in DSC Networks

A data aggregation query is converted to a web service composition before execution. This is an attractive solution compared to TCP/IP calls used by traditional sensor networks as this provides a more interoperable solution that is suitable for a distributed environment with heterogeneous camera nodes. In this section we discuss in detail, the process of converting SQL queries to web service compositions and how the composition is executed.

3.3.1 Mapping SQL Queries to Web Service Compositions

In application specific environments like smart camera networks, there is usually a fixed set of queries executed such as, detecting faces, finding aggregate of detected number of faces, etc. Therefore, a static process mode is more suitable for a smart camera environment compared to a dynamic service composition model. Therefore, for each query offered by the network, the web service composition is already predefined. A simplified version of the distributed web service composition algorithm is given below:

Algorithm1: Web Service Composition

```
Input:
ws              :the web service invoked by my parent
ws.getData()    :the method  of ws invoked by my parent
duration        :the duration for image readings/face detection
period          :the sample period for image readings/face detection


Steps:
1.    Create three threads : ThreadA, ThreadB, ThreadC
2.    Execute threads by calling simultaneously:
      2.1.  ExecuteThreadC(BlockingQueue)
            2.1.1         For each element e in BlockingQueue
                  2.1.1.1 Invoke remote ws.putData(e) of my parent
      2.2.  ExecuteThreadA(BlockingQueue, duration, frequency)
                  2.2.1.    Int numFaces = Invoke local FaceDetector.getFaces()
                  2.2.2.    BlockingQueue.EnQueue(numFaces)
      2.3.  ExecuteThreadB()
                   2.3.1.  For ChidNodeId = 0 to n do:
                           2.3.1.1  Invoke ws.getData()
```

This distributed algorithm is executed by each camera node upon invocation of one of its local web service for the requested query by its parent. This algorithm makes use of both web services and local services of the camera node. This distributed algorithm is designed and implemented for a basic query execution and can easily be extend to aggregate query execution and core services execution via queries. As shown by the algorithm, this composition is triggered when a local web service (*ws)* (e.g. MaxService) *getData()* method of a camera node(denoted by *ws.getData()* in algorithm) is invoked by its parent node. Upon this invocation, the camera node simultaneously deploy three threads: one that invoke the local face detector service to read images, one to invoke children nodes *ws.getData()* method to propagate the service request (query) to children and another to invoke *ws.putData()* of the parent node, to send the results back to the parent. The results generated by local *FaceDetector* component and the result sent by children nodes (by using *ws.putData()* of the local web service *ws*) are placed in a blocking queue called *BlockingQueue*. The operation of the blocking queue is discussed in more depth in chapter 5. We also left out the details of a camera node using *ParentClient* and *ChildClient* services for communication for simplicity.

Example Web Service Composition

Figure 3.9 depicts an example web service composition scenario when the *getData()* method of service *MaxService* of node 1 (denoted as *MaxService.getData()* in Figure 3.8) is invoked by its parent. The interactions shown in the figure are only for one parent, child pair. Upon invocation of its local web service *MaxService*, parent camera node (node 1) simultaneously starts to invoke its local *FaceDetector* service *getData()* method to start collecting image readings at specified intervals for the given duration, invoke *MaxService.getData()* service method of all of its children (figure shows only one child node, node 1) and send results back to its own parent by calling *MaxService.putData()* service method of the parent node whenever it gets results (i.e. maximum number of faces) to its blocking queue either from its local services or children services.

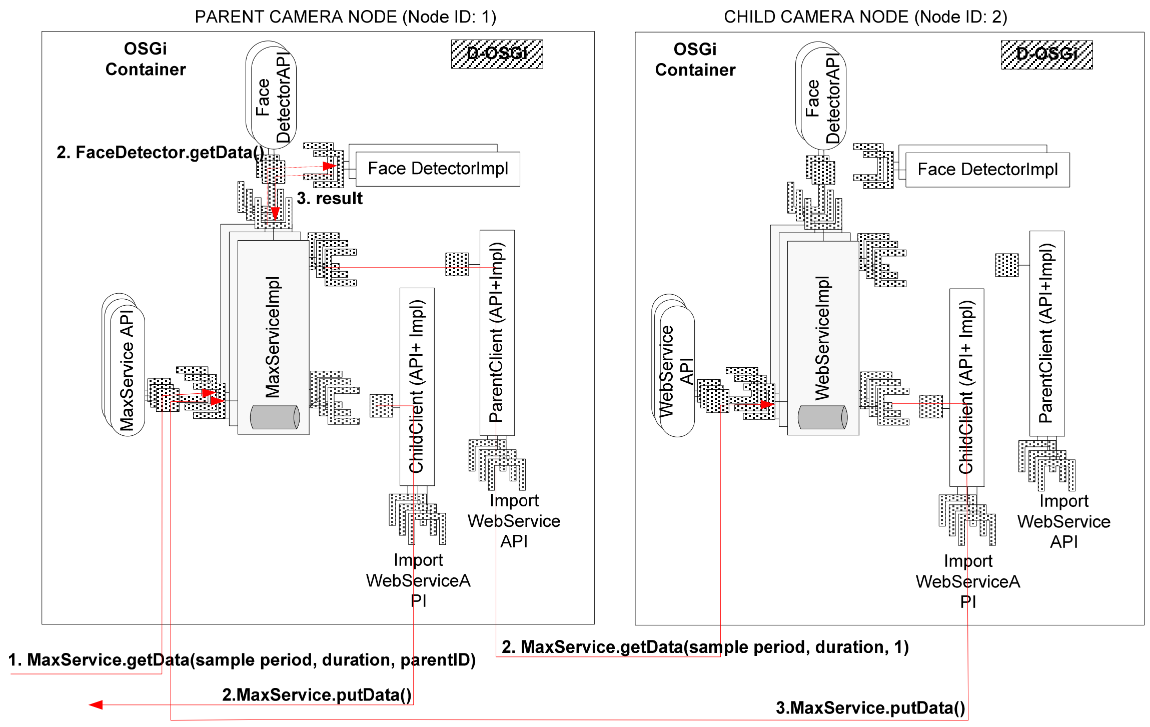


Figure 3:8 Example Web Service Composition for MaxService

3.3.2 Producer-Consumer Asynchronous Communication Architecture

When a query is executed, the query request is propagated from the root node to entire sub-tree rooted at it by forwarding the query down the tree from parents to their children. When the results are collected by camera nodes, they are propagated back to the root node from children to their parents. This query propagation from parent node to a child node takes place as remote (i.e. child node) web service invocation from the parent node. For example, the aggregate query for finding maximum number of faces detected in a given duration will result in each parent node receiving the query invoking the remote *MaxService* of its children.

Using typical server client web service invocation for allowing a parent to call its children fails in our network because these operations (i.e. web services) are too time consuming. For example, assume a camera node (client) N has three children (servers) and the node invoke remote web service of each child node sequentially using the traditional server client web service invocation method. Then once N invokes the remote web service of one child it must wait until this child node send results back, before it invoke the remote web service of the next child. This wait time is usually very long as once the remote web service of child node is invoked, each child will invoke remote web service of its own children and wait on them to return results and so on. This result in N waiting until all the camera nodes in the sub-tree rooted at it return results. This problem is further amplified depending on the time taken by an operation, the user query parameters (for example sample rate), performance of the image processing algorithm, the depth of the hierarchical network and the type of operation performed on the node by the web service.

Solution to this problem is using asynchronous web service invocation when parent node invokes a remote web service of a child node, as this avoids parent camera node waiting till a child node return results, saving valuable time. One approach to solve this problem is to allow the parent camera node to cerate a new thread per child camera node to invoke the remote web service. This will allow the parent camera node to invoke remote web services of all child camera nodes at the same time, by deploying all threads simultaneously. However, the parent camera node still needs to wait till all child camera nodes return results. Therefore this approach is also not suitable for our work, as HTTP requests has a fixed wait time defined for expected reply messages (For example in Apache CXF default time out is 30 seconds). If response is not arrived in pre defined time HTTP client will release the resource and time out the connection. There is no way to guarantee that all children camera nodes of the parent camera node will return results before the connection time outs.

Therefore to solve this problem, we propose novel communication architecture between a parent camera node and a child camera node. Our approach is to allow both parent camera node and child camera node to act as both client and server. As shown in Figure 3.9, the parent camera node acts as client and invokes the remote web service in the child camera node which is the server. Upon invocation of its local web service by the parent camera node, the child camera node immediately returns a dummy reply to the parent camera node by creating worker threads in the child node. This will stop the indefinite wait of the parent camera node on child camera node results and parent camera node can engage in other useful activities. When the result become available in the child camera node it will this time act as the client and invoke web service in the parent camera node (this time, the server) with the result. We call this asynchronous communication architecture the *producer consumer architecture* where the parent is the consumer and the child is the producer.

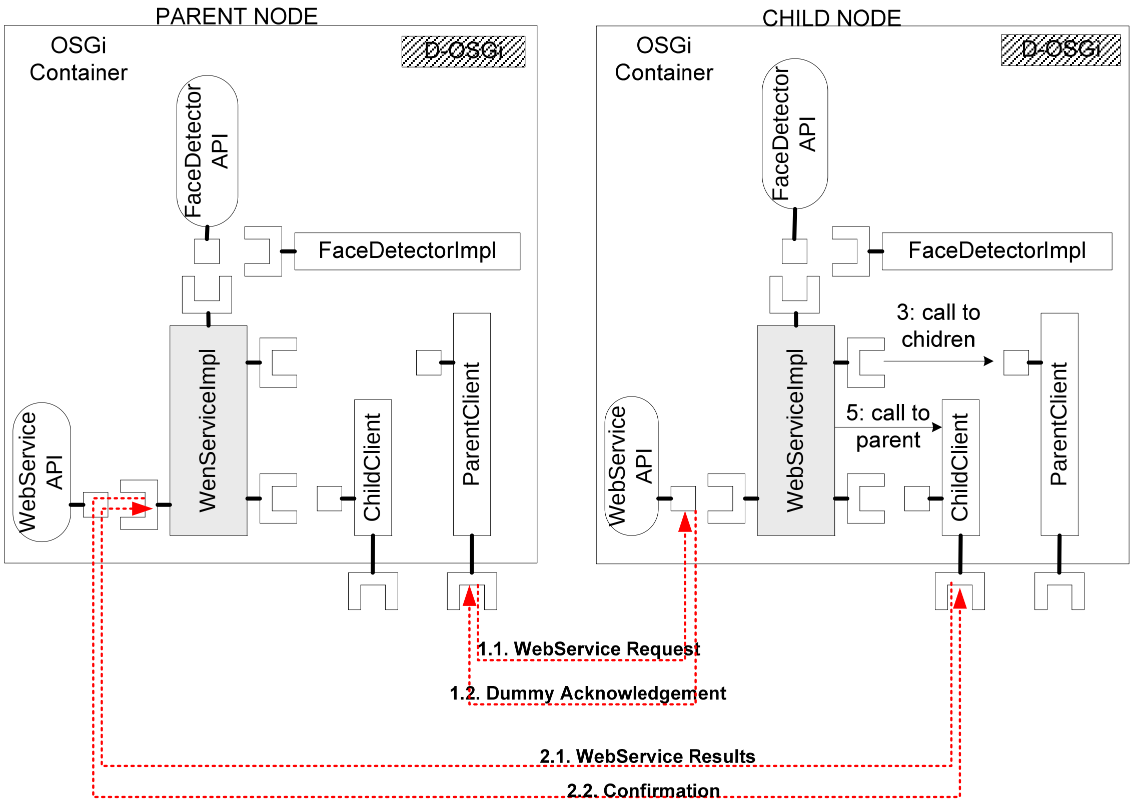


Figure 3:9 Producer-Consumer Asynchronous Communication Architecture

4 IMLEMENTATION AND PERFORMANCE EVALUATION

4.1 Implementation

4.1.1 System Parameters

In our work, we simulated smart camera network with average 10 camera nodes. Each node has a degree ranging from 2 to 3. The camera nodes are organized into the hierarchical network architecture illustrated in Figure 4.1. Two critical parameters: processing power and battery life is used for deciding the number of child nodes a camera node can have. In this work, we implemented up to three levels of camera nodes in the hierarchy. Every camera node is run on Apache Felix which is a commercially available open source OSGi container. As distributed OSGi, Apache cxf-dosgi is then installed on top of Apache Felix framework in every node. This work can easily be ported into real smart camera nodes. Table 4.1 summarizes these system parameters.

Table 4:1 Network Parameters and System software

System Parameter	Software Platform/ Algorithm/Value
Bundle Development	Eclipse Plug-in Development Environment
Root Camera Node Container	Eclipse Equinox OSGi Container
Intermediate and Leaf Camera Node Containers	Apache Felix OSGi Containers
HTTP Server	Jetty
Web Service Development Environment	Apache cxf dosgi (D-OSGi)
Face Detection Algorithm	ViolaJones Face Detection Algorithm
Smart Camera Network Architecture	Hierarchical Network
Network Size	Depth from root (Maximum 3), Number of nodes (Maximum 15)
Average Degree of a Smart Camera Node	2-3

4.1.2 Implementation Issues

We encountered several implementation issues in this research project, which we discuss below.

(i)Handling Streaming Data

Unlike traditional computer networks, DSC networks produce data streams. These data streams can either be raw image data streams, or aggregates over some attribute of the image streams. Such behavior is very useful for surveillance and other monitoring applications which will monitor the network behavior over the time. However, image processing in the smart camera is a time consuming process and this processing is carried out at specified sample intervals. Also, a camera node has to wait for its own local reading and reading from its children camera nodes to become available to send the data up in network towards root node. To enable this streaming behavior, we use a data structure called *blocking queue*. Blocking queue is a queue that will wait if you are trying retrieving an element when the queue is empty. A camera node loops through a blocking queue until all the data are processed.

Blocking queue is used by both local services and remote web services to put results. For example, when the local web service *MaxSercvice* of a node is invoked, it in turn invokes the remote *MaxService* of children nodes and local *FaceDetector* service. Both local FaceDetector and the remote children node *MaxService* will place multiple data items in camera node's blocking queue. Number of data items placed by the local *FaceDetector* depends on the sample period and the duration. Number of data items placed by remote *MaxService* depends on number of children camera nodes it has the sub-tree rooted at itself. Each camera node in the sub-tree will send one data Item (maximum found for the camera node) each. In case this web service was not an aggregate, but a simple *RawDataService*, then each of these nodes will send multiple data items instead. While propagating results back to the parent in the hierarchy, each node needs to find the local maximum number of faces. To achieve this, we associate a status tag for each data item entered in to the blocking queue. Value zero in this status tag indicates that more data needs to arrive from the same source, while a value one in the status tag indicates that it is the final data item from the given source. When processing elements in the blocking queue, the *MaxSerivice* first checks the status tag of data and calculate and send results to its parent camera node.

(ii) Handling Lengthy Operations

We introduced our proposed producer consumer architecture on top of distributed OSGi in chapter 4. The soap messages passed between the consumer and the producer to invoke the producer for a lengthy operation and the dummy reply as tracked by Local Network Monitor [77] network monitoring software look like as follows.

a. Invoking Service Asynchronously

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
      <soap:Body>
            <ns1:getStat xmlns:ns1="http://childserviceinf/">
                  <ns1:arg0>2</ns1:arg0>
                  <ns1:arg1>20</ns1:arg1>
                  <ns1:arg2>0</ns1:arg2>
            </ns1:getStat>
      </soap:Body>
</soap:Envelope>
```

b. Dummy reply

```
HTTP/1.1 200 OK..Content-Type: text/xml; charset=utf-8..Content-Length:
168..Server: Jetty(6.1.x)....
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
      <soap:Body><ns1:getStatResponse xmlns:ns1="http://childserviceinf/" />
      </soap:Body>
</soap:Envelope>
```

As discussed in chapter 4, we devise a new Producer/Consumer communication architecture to handle the situation.

(iii) Types of Messages

Types of messages passed between the camera nodes depend on the query issued by the user. Messages can carry either aggregated or raw data. Raw data are the data that is not subjected to aggregation but been subjected to some processing such as detecting the number of faces of an image reading. These data will form a temporary table on each camera node that participates in the query.

Figure 4.1 depicts the types of message formats that are used in our smart camera network. Figure 4.1 (a) shows the message format for select and aggregate queries. *Sample rate* and *duration* are user specified parameters and represent the image reading interval and the duration of sample collection respectively. Each reading taken as a result of the query receipt will be subjected to face detection process later on. As shown in Figure 4.1 (b), a temporal aggregate query allows the user to specify two other key input parameters: the *response rate* and the *window size*. Response rate specifies the interval the camera node must send response back to the user and the window size specified the time period from the current response time that needs to be used to calculate the response value. Figure 4.1 (c) shows the response message format for queries. There is only one response message format for all queries with minor exceptions. For a select and aggregate query, *epoch* represent the sample time interval for which the response was generated. This epoch filed is replaced by the filed *step* in a response message for a temporal aggregate query. This is the response time interval for which the response was generated. The *NumFaces* filed for a select query contains the detected number of faces at that given epoch at that node while an aggregate and temporal aggregate queries, this filed contains the requested aggregate of the NumFaces filed. In response to a query, a camera node can generate multiple response messages. For example, for a select query with sample period for 5 seconds and duration 15 seconds, 4 response messages are generated, one at each sample interval. It is crucial for the camera node to identify which response message is the last response message for a given query, to stop executing the query further and start calculating aggregate if the query was an aggregate or temporal aggregate query. For this purpose, a camera node generating a response message attaches a *status* bit to represent if this is the last response message for the query or not by setting status bit to 1 or 0 respectively.

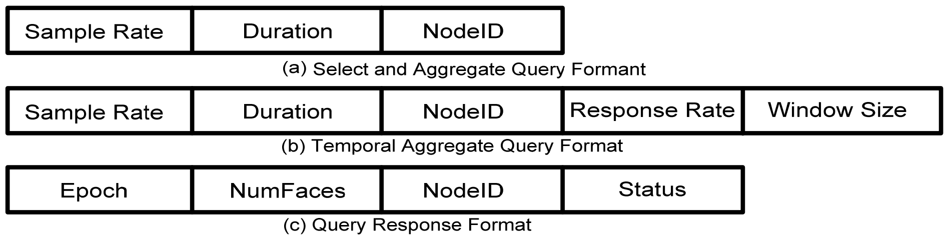


Figure 4:1 Standard Message Formats

NodeID is the node id of the message sender camera node. For select, aggregate and temporal aggregate queries, this is usually the parent node id of the query message receiving camera node. For a select response message, the NodeID is the id of the node that collected the image data for which the number of faces (numFaces) is included in the message. NodeID of an aggregate response message has two cases. First, if the aggregate is average, the response message contains the node id of the final sender of the response message (the root camera node) in the NodeID filed. Second, if the aggregate is a non-average aggregate such as min or max, the NodeID contains the id of the camera node which detected the maximum or minimum number of faces in the network respectively.

4.2 Results

4.2.1 Performance Metrics

We measure the performance of our system using following metrics:

(i) Turnaround Time

Turnaround time is the time taken between the query issue at root and the arrival of the last response message for that query back to the root. This measures the communication cost in terms of time.

(ii) Bytes Transferred

This is the total number of bytes transferred within the network for a given query. This measures the communication cost in terms of bandwidth consumption. This also is an indication of the effectiveness of the load balancing strategy.

(iii) Service Startup Time

Service startup time is the time taken between the service startup request triggered by a query issue at the root node and the arrival of the confirmation of successful startup of all camera nodes in the network at the root.

(iv) Memory Consumption

This shows the consumption of the memory of a single camera node as a result of running services.

4.2.2 Experimental Results And Performance Analysis

We collected the results using LocalNetworkMonitor 3.1 [77] network monitoring software. Each camera node in our implementation uses a unique port. The network monitoring software captures the messages transferred between these ports. Below we present the results of the experiments we conducted.

(i) Turnaround Time

Figure 4.2 shows the turnaround time for the SELECT operation. The duration is set to 20 seconds and we experimented for sample intervals from 1 second to 5 seconds for different sized networks: depth 1 for a 3 nodes network, depth 2 for a 7 nodes network and depth 3 for a 15 nodes network. As shown in the graph, for a given sized network, the turnaround time increases with a lower sample interval. This is usually expected because when sample interval is low more data are generated by each node during the fixed time interval. This results in congestion at the root node due to traffic increasing the turnaround time. This increase in turnaround time is more apparent in larger network sizes because the network depth is higher in these networks. Therefore messages need to travel longer distances to reach camera nodes.

Figure 4.3 depicts the turnaround time for the MAX aggregate operation. Similar to the previous experiment, the duration is fixed to 20 seconds. Same pattern of behaviour observed for SELECT in Figure 4.2 can be observed here for similar reasons. However, as can be observed, the turnaround time is less for MAX than for SELECT operation. This change again is more obvious for larger network sizes. The reason for this is that due to in-node processing (aggregation), the data traveling from camera nodes toward the root will significantly reduce with lower depths from the root.

Figure 4.4 depicts the turnaround time for temporal aggregate WINMAX. Here we fix the duration to 20 seconds and sample rate to 1 second and experiment the turnaround time for different response intervals (I second to 5 seconds) for different network sizes. Higher turnaround times can be observed when response interval is 1 second. The reason for this is that lower response intervals results in higher congestion in the network.

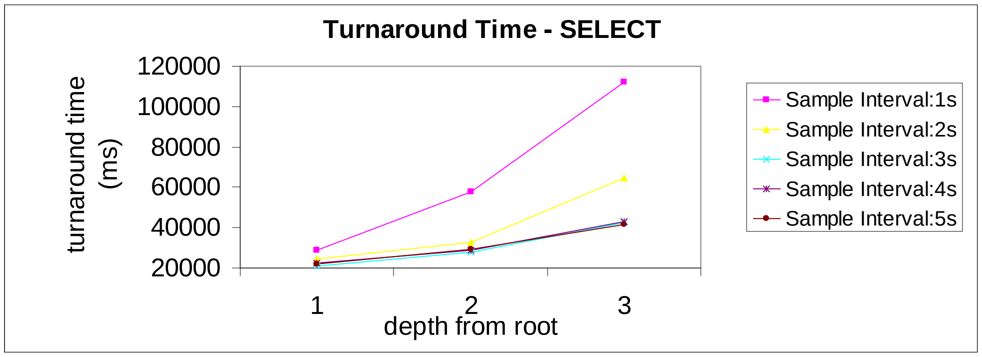


Figure 4:2 Turnaround Time for SELECT operation

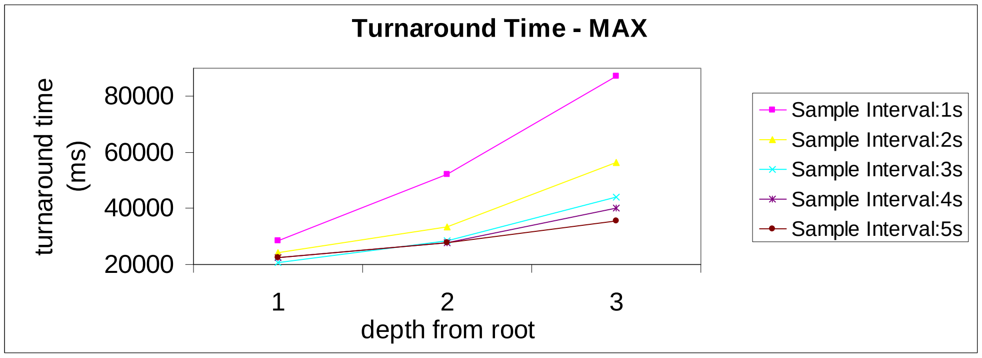


Figure 4:3 Turnaround Time for MAX operation

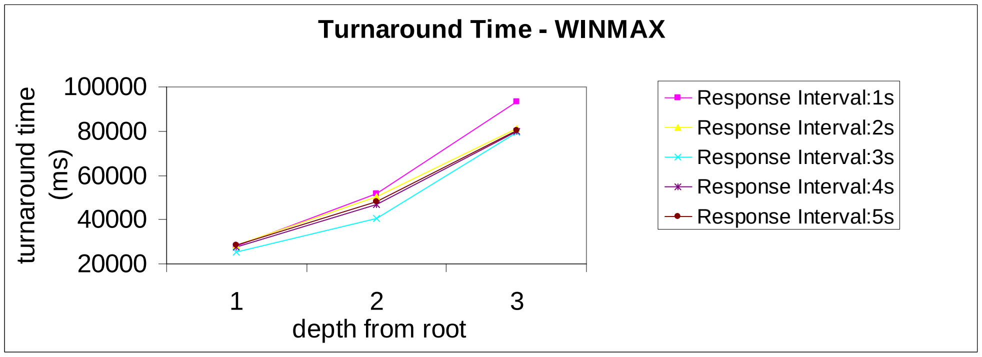


Figure 4:4 Turnaround Time for WINMAX operation

(ii) Total Bytes Transferred

Figure 4.5 shows the bytes transferred per epoch for aggregation MAX in a 15 nodes network, for a duration of 20 seconds and sample interval of 1 second. We compare our hierarchical aggregation scheme with a flat network architecture, where all the camera nodes directly send raw data to the sink without performing aggregation (sink performs aggregation). As can be expected, our hierarchical aggregation scheme outperforms flat network aggregation counterpart by 55.40%. The huge reduction in total bytes transferred during the aggregation operation results from the in-node local aggregation occurring at every node. Our scheme therefore distribute load more fairly than flat network aggregation.

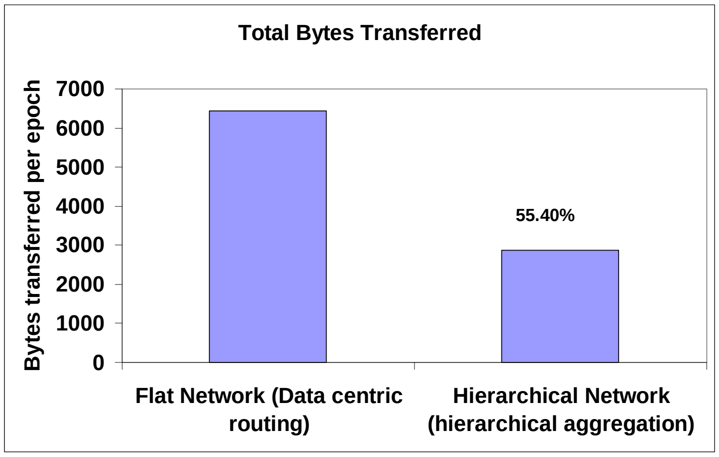


Figure 4:5 Total Bytes Transferred

(iii) Service Startup Time

Figure 4.6 shows the service startup times for different sized networks. Higher number of nodes and more levels in the hierarchy results in service startup request to travel to many nodes at longer distances. This essentially results in higher startup times for larger networks.

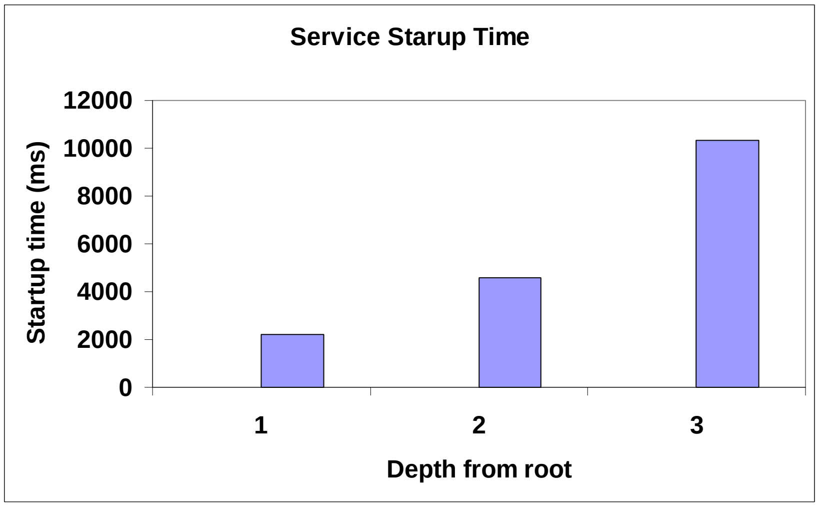


Figure 4:6 Service Startup Time

(iv) Memory Consumption

Figure 4.7 depicts the memory consumption for local services for a single camera node. Memory consumption is shown for different number of local services is running in the node. No web services are run during this experiment. The service we selected to run in this experiment is shown in Table 4.2.

Table 4:2 Local Services Selected to Run in a Smart Camera Node

Number of Local Services Running	Local Services Running
0	Empty OSGi Container
1	FaceDetector
2	FaceDetector, ChildClient
3	FaceDetector, ChildClient, ParentClient

The OSGi container which runs the camera node, itself consumes 25224 kilo Bytes without any services running. When the number of deployed local services increase there is considerable increase in memory consumption. The percentage increases with respect to empty OSGi Container are shown in Figure 4.7 on top of each bar.

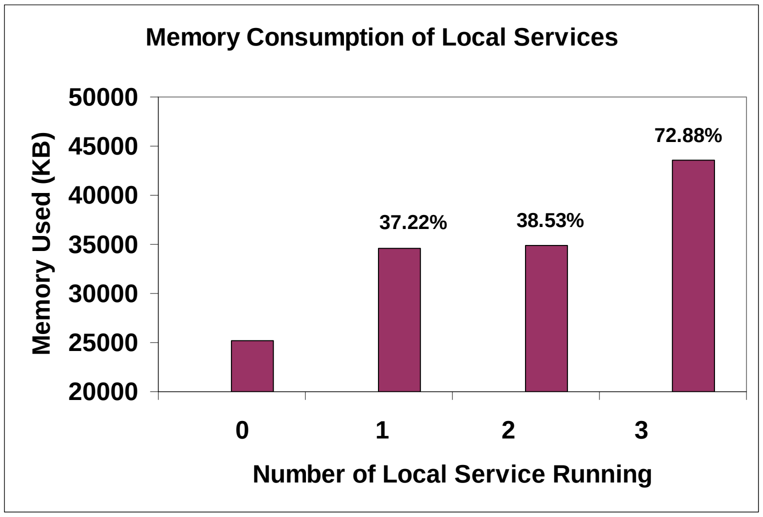


Figure 4:7 Memory Consumption of Local Services

Figure 4.8 depicts the memory consumption of a camera node when different number of web services running. We monitored the memory consumption for up to 8 web services running. The local services were running in this experiment. Table 4.3 shows the web services we selected to run in the camera node.

Table 4:3 Web Services Selected to Run in a Smart Camera Node

Number of Local Services Running	Local Services Running
1	Select
2	Select, Max
3	Select, Max, WinMax
4	Select, Max, WinMax, Avg
5	Select, Max, WinMax, Avg, Load
6	Select, Max, WinMax, Avg, Load, Min
7	Select, Max, WinMax, Avg, Load, Min, WinMin,
8	Select, Max, WinMax, Avg, Load, Min, WinMin, WinAvg

According to Figure 4.8 there is slight increase of memory consumption when more web services are running. The maximum increase we observed in this experiment was 4.56% (compared to memory consumption when only one web service is running) when 8 web service are running in the camera node.

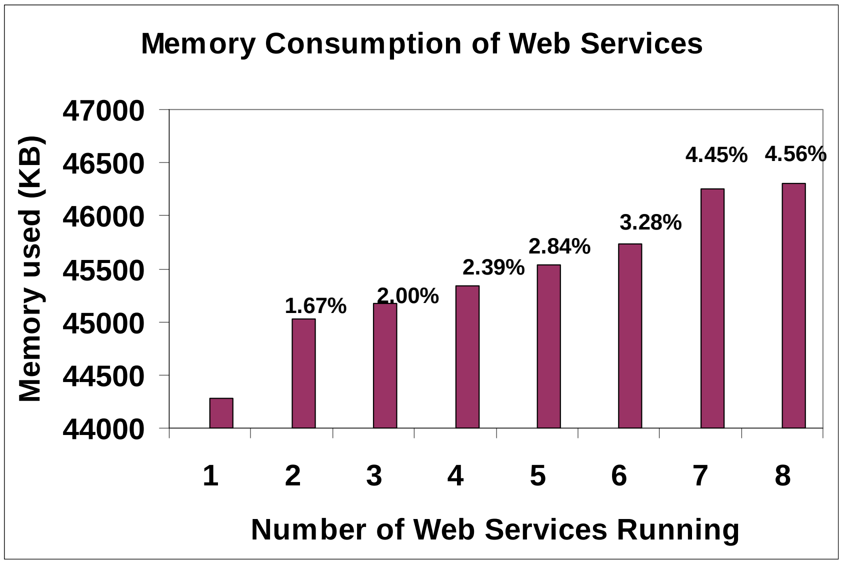


Figure 4:8 Memory Consumption of Web Services

Overall, we monitored that a camera node on average consumes 250 kilo bytes to run a single web ser-

vice.

5 CONCLUSION AND FUTURE WORK

5.1 Conclusion

We model and implement a web service enabled smart camera network that allows the user to do query processing. The users are provided with an SQL interface where they can issue queries to the network in a simple manner. The main focus of these newly developed SQL primitives were on aggregate queries. These queries are then internally converted to web service compositions, executed and return the result. The web service modeling using OSGi, web service composition, providing SQL API for sophisticated queries are our major contributions in this thesis work. To our knowledge none has answered these before. The web enabled DSC environment and the mapping of SQL queries to web service compositions relives the user from the burden of low level programming, while making camera node functionalities widely available to any user through web services.

5.2 Future Work

New Synchronization algorithms can be developed for the DSC network to allow collaborating nodes for a given query to start data collecting and processing at the same time. Also, in this thesis work we assume static hierarchical network architecture.

New algorithms need to be devised for on-the-fly creation of this hierarchical network. Core service can be implemented to form a routing tree in a wireless environment. This can be achieved by node setting the level and send the query to all and select lower level node as a parent when it hears the query from the routing tree. This will happen until every node organizes into a tree.

Enabling bundle (service) mobility in which a bundle (service) can migrate form one camera node to another will be a powerful tool for execution of distributed applications. The communication of proposed architecture entirely happens though web services. Web service enable environments pass only SOAP messages. Attaching a bundle to a SOAP message and installing it in specified OSGi container will be a challenging problem.

Extending the query API to do more sophisticated data analyses in monitoring applications is another area that needs to be addressed. The basic query types we provided for the smart camera network can be improved to allow developer to implement robust monitoring algorithms. By introducing queries for event detection in the network allow automatic detecting and initiating of responses which will be useful in monitoring application. By using the dynamic nature of the OSGi container and power of java programming, an application developer can introduce new type of queries that eventually helping in monitoring applications. Develop sophisticated data analysis algorithm on top of smart camera network will be one of our future work.

Also, developing SQL primitives that allow user to do resource monitoring is another area that is still not addressed by the research community. Other than monitoring queries, network health queries and actuation quires can be introduce to the existing query API. Face detection service in the given architecture can be easily replace with any other detector for example current flow detector, memory consumption detector etc. In network heath monitoring queries can use those detector services as needed. This will also helps to implement task allocation algorithms and load balancing algorithms.

Face detection is a difficult problem to many problems such as partially occluded, law resolution, lighting and head poses. The difficulties arises in such scenarios can be address with spatial and temporal information. If the children have overlapping regions then the parent can do more work in addition to aggregation. Children nodes can send partial result of face detection to their parent with spatial and temporal information then parent can decide the hypothesis of the existence of a face providing collaborative face detection algorithms.

Sensor fusion is another popular area of research. Cameras can be equipped with others types of sensors for example sound, temperature, IR etc. Information from these sensors can be used to solve ambiguity and uncertainty. However it is necessary to correlate these data to make decisions. Use of such algorithms will be helpful in monitoring algorithms.

6 REFERENCES

[1] Samuel R. Madden et al, Tinydb: An Acquisitional Query Processing System For Sensor Networks, 2005.

[2] Joseph M. Hellerstein et al, Beyond Average: Toward Sophisticated Sensing With Queries, 2003.

[3] J´Er ´Emie Leguay et al, An Efficient Service Oriented Architecture For Heterogeneous And Dynamic Wireless Sensor Networks, 2008.

[4] Junsuk Shin et al, ASAP: A Camera Sensor Network For Situation Awareness, 2007.

[5] Hamid Aghajan et al, Multi Camera Networks Principles And Applications, 2009.

[6] Nissanka B. Priyantha et al. Tiny Web Services: Design And Implementation Of Interoperable And Evolvable Sensor Networks, 2008.

[7] Martin Tsenov, Example Of Communication Between Distributed Network Systems Using Web Services. 2007.

[8] Paul Viola, Michael Jones, Robust Real-Time Object Detection, 2001

[9] Anthony Rowe, Adam Goode, Dhiraj Goel, Illah Nourbakhsh, Cmucam3: An Open Programmable Embedded Vision Sensor, Carnegie Mellon Robotics Institute Technical Report, RI-TR-07-13 May 2007.

[10] M. Jones, P. Viola, Fast Multi-View Face Detection, MERL, TR2003-96, July 2003.

[11] J. Nesvadba, A. Hanjalic, P. M. Fonseca1, B. Kroon, H. Celik, E. Hendriks, Towards A Real-Time And Distributed System For Face Detection, Pose Estimation And Face-Related Features , Int. Conf. On Methods And Techniques In Behavioral Research, 2005.

[12] Paolo Costa, Geoff Coulson et al ,The RUNES Middleware For Networked Embedded Systems And Its Application In A Disaster Management Scenario, IEEE International Conference On Pervasive Computing And Communications, 2007.

[13] D. Bellebia, J-M. Douin, Applying Patterns To Build A Lightweight Middleware For Embedded Systems, Conference On Pattern Languages Of Programs, 2006.

[14] Christopher Gill , Venkita Subramonian , Douglas Niehaus , Douglas Stuart , Jeff Parsons Huang-Ming Huang, ORB Middleware Evolution For Networked Embedded Systems, In Proceedings Of The 8th International Workshop On Object Oriented Real-Time Dependable Systems (WORDS'03) 2003.

[15] Panahi, M. Harmon, T. Klefstad, R., Adaptive Techniques For Minimizing Middleware Memory Footprint For Distributed, Real-Time, Embedded Systems, Dept. Of Electr. Eng. & Comput. Sci., California Univ., Irvine, CA, USA IEEE 18th Annual Workshop On Computer Communications, 2003.

[16] D. C. S. et al, TAO: A Pattern-Oriented Object Request Broker For Distributed Real-Time And Embedded Systems, IEEE Distributed Systems Online,Vol. 3, Feb. 2002.

[17] D. C. Schmidt, ACE: An Object-Oriented Framework For Developing Distributed Applications, In Proceedings Of The USENIX C+ + Technical Conference, (Cambridge, Massachusetts), USENIX Association, Apr. 1994.

[18] Object Management Group, Minimum CORBA - Jointvrevised Submission, OMG Document Orbos/98-08-04ved., Aug. 1998.

[19] Cheng Chen, Bin Tian, Ye Li And Qingming Yao, Data Aggregation Technologies Of Wireless Multimedia Sensor Networks:A Survey, 2010.

[20] A.S. Tanenbaum, M. Van Steen, Distributed Systems: Principles And Paradigms, Prentice Hall, 2006.

[21] I.F. Akyildiz, T. Melodia, K.R. Chowdhury, A Survey On Wireless Multimedia Sensor Networks, Computer Networks 51 (2007) 921–960 2007.

[22] Ramesh Rajagopalan And Pramod K. Varshney, Syracuse University, Data-Aggregation Techniques In Sensor Networks- A Survey, 2006

[23] Ambuj Shatdal And Jeffrey Naughton. Adaptive Parallel Aggregation Algorithms. In ACM SIGMOD, 1995

[24] Weipeng P. Yan And Per Ake Larson. Eager Aggregation And Lazy Aggregation. In VLDB, 1995.

[25] Latha Srinivasan And Jem Treadwell HP Software Global Business Unit, An Overview Of Service-Oriented Architecture, Web Services And Grid Computing, 2005

[26] Doulkeridis, C., Valavanis, E. And Vazirgiannis, M. Benatallah, B. And Shan, M-C. (Eds.): Towards A Context-Aware Service Directory, TES, LNCS 2819, Springer-Verlag Berlin Heidelberg, Pp.54–65 2003.

[27] D. Martin Et Al. DAML-S(And OWL-S) 0.9 Draft Release. Online: http://www.Daml.Org/Services/Daml-S/0.9/, May 2003.

[28] Jinghai Rao And Xiaomeng Su, A Survey Of Automated Web Service Composition Methods, 2004.

[29] Dimka Karastoyanova, Alejandro Buchmann, Components, Middleware And Web Services, 2003.

[30] Intanagonwiwat C, Govindan R, and Estrin d, Directed Diffusion: A Scalable And Robust Communication Paradigm For Sensor Networks. In *Mobicom*. Boston, MA 2000,
[31] Madden S, and Franklin M J, Fjording, The Stream: An Architechture For Queries Over Streaming Sensor Data. In ICDE, 2002.
[32] Yao Y, and Gehrke J, The Cougar Approach To In-Network Query Processing In Sensor Networks. In SIG-MOD Record. 2002.
[33] Fatih Emekci, Hailing Yu, Divyakant Agrawal, And Amr El Abbadi, Power-Aware Query Processing Over Sensor Networks, 2003.
[34] Nalla Senthilnathan , Develop And Deploy Web Services As Osgi Bundles, 2009.
[35] Rahmat Bagas Santoso, Distributed Osgi Through Web Services, 2009.
[36] Paul Viola, Michael J. Jones , Robust Real-Time Face Detection,2004.
[37] http://cxf.apache.org/dosgi-releases.html
[38] Craig Walls, Modular Java, Creating Flexible Applications with OSGi and Spring, 2009.
[39] Philippe Bonnet, Johannes Gehrke, Praveen Seshadri, Towards Sensor Database Systems, 2001.
[40] Chien-Chung Shen, Chavalit Srisathapornphat, and Chaiporn Jaikaeo, Sensor Information Networking Architecture and Applications, 2001.
[41] J. Kulik, W. R. Heinzelman, and H. Balakrishnan, Negotiationbased Protocols for Disseminating Information in Wireless Sensor Networks, Wireless Networks, vol. 8, Mar. 2002, pp. 169–85.
[42] C. Intanagonwiwat, R. Govindan, and D. Estrin, Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks, Proc. 6th Annual Int’l. Conf. Mobile Comp. and Net. (MobiCOM ’00), Aug. 2000.
[43] B. Krishnamachari and J. Heidemann, Application Specific Modeling of Information Routing in Wireless Sensor Networks, Proc. IEEE Int’l. Performance, Computing and Commun. Conf., vol. 23, 2004, pp. 717–22.
[44] W. R. Heinzelman, Application-Specific Protocol Architectures for Wireless Networks, Ph.D. thesis, Massachusetts Institute of Technology, June 2000.
[45] O. Younis and S. Fahmy, HEED: a Hybrid, Energy-Efficient, Distributed Clustering Approach for Ad Hoc Sensor networks, IEEE Trans. Mobile Computing, vol. 3, no. 4, Dec. 2004, pp. 366–79.
[46] S. Lindsey, C. Raghavendra, and K. M. Sivalingam, Data Gathering Algorithms in Sensor Networks Using Energy metrics, IEEE Trans. Parallel and Distributed Systems, vol. 13, no. 9, Sept. 2002, pp. 924–35.
[47] M. Ding, X. Cheng and G. Xue, Aggregation Tree Construction in Sensor Networks, 2003 IEEE 58th Vehic. Tech. Conf., vol. 4, no. 4, Oct. 2003, pp. 2168–72.
[48] J. Llinas, D.L. Hall, An introduction to multi-sensor data fusion, in: Proceedings of the IEEE International Symposium on Circuits and Systems, 1998.
[49] D.L. Hall, J. Llinas, Handbook of Multisensor Data Fusion, CRC Press, 2001.
[50] Papazoglou, M. P., van den Heuvel, W., Service oriented architectures: approaches, technologies and research issues, The VLDB Journal, vol. 16, no. 3, July 2007, pp. 389-415
[51] Bronsted, J., Hansen, K. M., Ingstrup, M., A survey of service composition mechanisms in ubiquitous computing, in Proc. UbiComp 2007 Workshop, pp. 87-92.
[52] Sirin, E., Parsia, B., Hendler, J., Composition-driven Filtering and Selection of Semantic Web Services, In AAAI Spring Symposium on Semantic Web Services, 2004.
[53] Sirin, E., Hendler, J., Parsia, B., Semi-automatic Composition of Web Services using Semantic Descriptions, In Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003, Angers, France, April 2003.
[54] http://www.w3.org/Submission/OWL-S/
[55] Whitehouse, K., Zhao, F., Liu, J., Semantic Streams: a Framework for Composable Semantic Interpretation of Sensor Data, EWSN 2006.
[56] Wang, X., Wang, J., Zheng, Z., Xu, Y., Yang, M., Service Composition in Service-Oriented Wireless Sensor Networks with Persistent Queries, Consumer Communications and Networking Conference, CCNC 2009.
[57] Bamis, A., Singh, N., Savvides, A., An Architecture for Dynamic Reconfiguration of Data Flows in Sensor Networks, Technical Report, ENALAB, Yale University, 2007.
[58] Bakshi, A., Prasanna, V. K., Reich, J., Larner, D., The Abstract Task Graph: A methodology for architecture independent programming of networked sensor systems, in Proc. Workshop on End-to-end, sense-and-respond systems, applications and services, 2005.
[59] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In ACM SIGMOD 2003, June 2003.
[60] Yong Yao and Johannes Gehrke. Query processing for sensor networks. In CIDR 2003, January 2003.

[61] H. Schneiderman and T. Kanade, A statistical approach to 3d object detection applied to faces and cars, IEEE Conference on Computer Vision and Pattern Recognition, 2000.

[62] Henry A. Rowley, Shumeet Baluja, and Takeo Kanade, Neural networkbased face detection, IEEE Transactions on Pattern Analysis and Machine Intelligence 20 (1998), no. 1, 23-38.

[63] C. P. Papageorgiou, M. Oren, and T. Poggio, A general framework for object detection, Proceedings of International Conference on Computer Vision (1998), 555-562.

[64] Ardizzone, E.; La Cascia, M.; Morana, M, Face Processing on Low-Power Devices, Embedded and Multimedia Computing, 2009. EM-Com 2009. 4th International Conference on 2009

[65] T. Yan, D. Ganesan, and R. Manmatha, Distributed image search in camera sensor networks, in SenSys ’08: Proceedings of the 6th ACM conference on Embedded network sensor systems. New York, NY, USA: ACM, 2008, pp. 155–168

[66] D. Xie, T. Yan, D. Ganesan, and A. Hanson, Design and implementation of a dual-camera wireless sensor network for object retrieval, in IPSN ’08: Proceedings of the 7th international conference on Information processing in sensor networks. Washington, DC, USA: IEEE Computer Society, 2008, pp. 469–480

[67] A. Rowe, A. G. Goode, D. Goel, and I. Nourbakhsh, Cmucam3: An open programmable embedded vision sensor, Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-07-13, May 2007.

[68] K. Khattab, J. Mitteran, J. Dubois, and J. Matas, Embedded system study for real time boosting based face detection, IEEE Industrial Electronics, IECON 2006 - 32nd Annual Conference on, pp. 3461–3465, Nov. 2006.

[69] http://www.cmucam.org/

[70] http://www.cmucam.org/wiki/viola-jones

[71] http://www.tinyos.net/

[72] Constantin Timm, Jens Schmutzler Peter Marwedel, Christian Wietfeld, Dynamic Web Service Orchestration applied to the Device Profile for Web Services in Hierarchical Networks, 2009.

[73] D.C. Schmidt, Middleware for real-time and embedded systems, Communications of theACM 45 (6) (2002) 43–48 2002.

[74] Y. Yu, B. Krishnamachari, V.K. Prasanna, Issues in designing middleware for wireless sensor networks, IEEE Network 18 (1) (2004) 15–21.

[75] M.M. Molla, S.I. Ahamed, A survey of middleware for sensor network and challenges, in: Proceedings of the IEEE International Conference on Parallel Processing,Workshops, 2006.

[76] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K.Whitehouse,A.Woo, D. Gay, J. Hill, M.Welsh, E. Brewer, D. Culler, Tinyos: An operating system for sensor networks, in: Ambient Intelligence, pp. 115–148, Springer, 2005

[77] http://www.ntkernel.com/

[78] M. Bramberger,A. Doblander,A.Maier, B. Rinner, H. Schwabach, Distributed embedded smart cameras for surveillance applications, Computer 39 (2) (2006) 68–75.

[79] S. Fleck, F. Busch,W. Strasser, Adaptive probabilistic tracking embedded in smart cameras for distributed surveillance in a 3D model, EURASIP Journal on Embedded Systems 2 (2007) 17.

[80] Y. Shi, T. Tsui, An FPGA-based smart camera for gesture recognition in HCI applications, Computer Vision: ACCV (2007) 718–727.

[81] E. Norouznezhad, A. Bigdeli, A. Postula, B.C. Lovell, A high resolution smart camera with GigE-vision extension for surveillance applications, in: Proceedings of the Second ACM/IEEE International Conference on Distributed Smart Cameras, 2008.

7 APPENDIX: System Requirements and Configuration

This will describe develop and deploy a cxf-dosgi service bundle and service consumer running in different JVMs. cxf-dosgi web service will deploy in Apache Felix and consumer bundle will deploy in Apache Equinox. We will use Eclipse for develop all the bundles and export as jar files.

7.1 Prerequisites

The following software is needed to install this framework:

1. OSGi : Apache Felix, Apache Equinox
2. Distributed OSGi : Apache cxf-dosgi
3. JDK 1.6
4. Eclipse JEE

7.2 Installation of Required Software

1. Install java 1.6

-Unpack the jdk installation into the required location

-Set the path variable to bin directory

2. Install eclipse-jee-galileo-SR2-win32.zip

- Download site:

http://www.gtlib.gatech.edu/pub/eclipse/technology/epp/downloads/release/galileo/SR2/

- Unpack the eclipse installation into the required location

3. Download cxf-dosgi single bundle distribution and osgi compendium bundle to a local directory.

- download site:

http://cxf.apache.org/dosgi-releases.html or http://archive.apache.org/dist/felix/org.osgi.compendium-1.2.0.jar

4. Install org.apache.felix.main.distribution-3.0.2.zip

- Download site

http://felix.apache.org/site/downloads.cgi

- Unpack the felix installation into the required location

7.3 Configuring Eclipse OSGi Container

The refers the article at http://www.ibm.com/developerworks/webservices/library/ws-OSGi/index.html

1. Start the OSGi container and register cxf-dosgi bundle as service provider enabler.

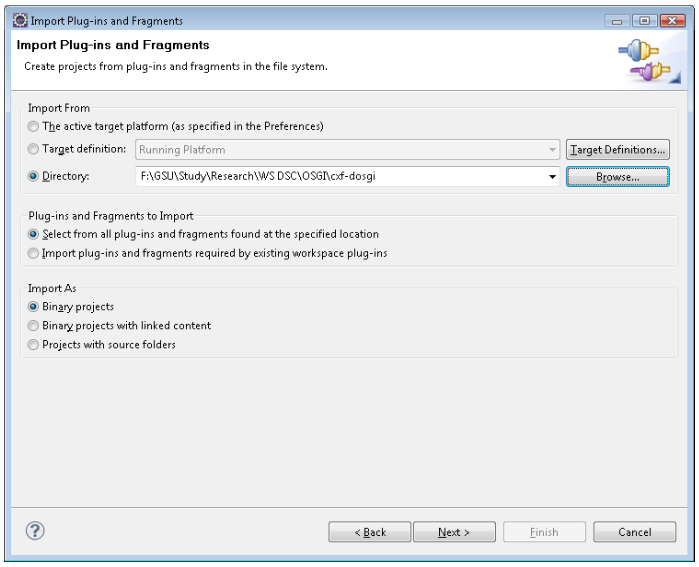
- Open empty work space in Eclipse

- Set the perspective to ‘plugin-development’

window->open perspective>other->select Plugin-Development

2. Import the cxf-dosgi single bundle distribution and osgi compendium bundle

File->import-> Plug-In Development->Plug-ins and Fragments -> select the directory where the bundles are located


Import Plug-ins and Fragments

Import Plug-ins and Fragments

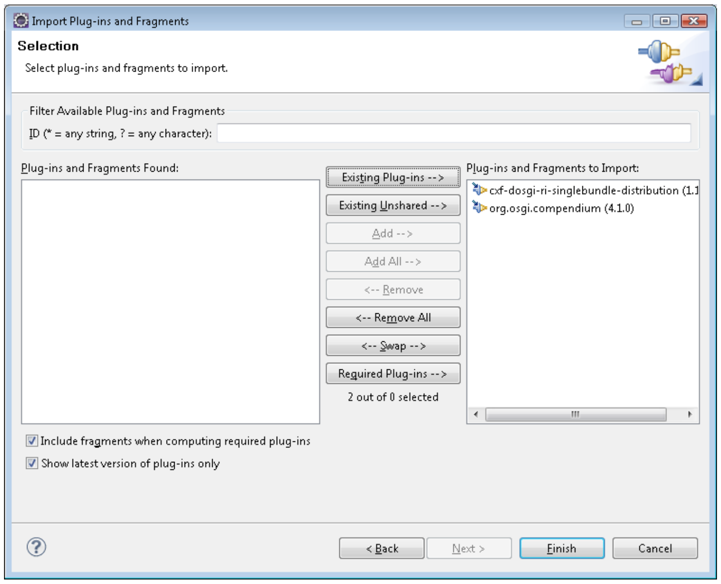
Create projects from plug-ins and fragments in the file system.

Import From
- ◯ The active target platform (as specified in the Preferences)
- ◯ Target definition: Running Platform [Target Definitions...]
- ● Directory: F:\GSU\Study\Research\WS DSC\OSGI\cxf-dosgi [Browse...]

Plug-ins and Fragments to Import
- ● Select from all plug-ins and fragments found at the specified location
- ◯ Import plug-ins and fragments required by existing workspace plug-ins

Import As
- ● Binary projects
- ◯ Binary projects with linked content
- ◯ Projects with source folders

3. click next -> add all -> finish


Import Plug-ins and Fragments

Selection

Select plug-ins and fragments to import.

Filter Available Plug-ins and Fragments

ID (* = any string, ? = any character):

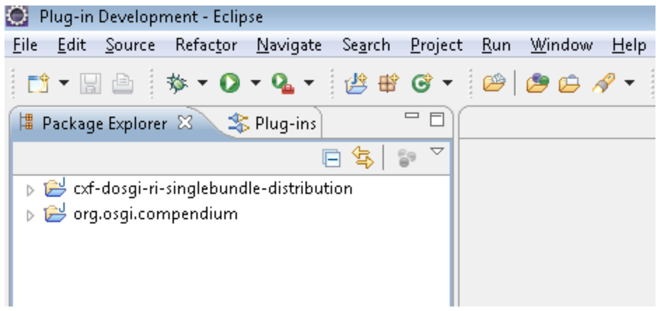
Plug-ins and Fragments Found:

Plug-ins and Fragments to Import:
- cxf-dosgi-ri-singlebundle-distribution (1.1
- org.osgi.compendium (4.1.0)

2 out of 0 selected

- ☑ Include fragments when computing required plug-ins
- ☑ Show latest version of plug-ins only

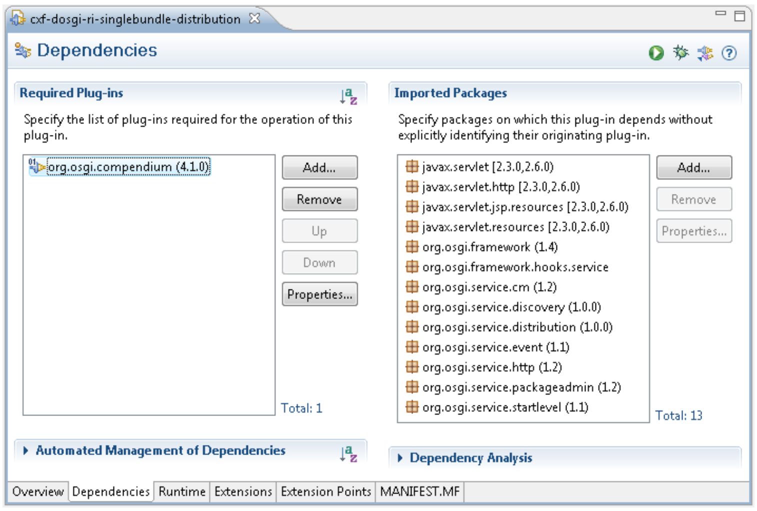
- This will create two plug-in development projects


Plug-in Development - Eclipse

- cxf-dosgi-ri-singlebundle-distribution
- org.osgi.compendium

4. Set dependency of cxf-dosgi-ri-singlebundle-distribution dundle

- double click on META-INF/MANIFEST.MF -> click on Dependencies tab -> org.osgi.compendium bundle as a required bundle



cxf-dosgi-ri-singlebundle-distribution

Dependencies

Required Plug-ins

Specify the list of plug-ins required for the operation of this plug-in.

org.osgi.compendium (4.1.0)

Add... Remove Up Down Properties...

Total: 1

Imported Packages

Specify packages on which this plug-in depends without explicitly identifying their originating plug-in.

javax.servlet [2.3.0,2.6.0)
javax.servlet.http [2.3.0,2.6.0)
javax.servlet.jsp.resources [2.3.0,2.6.0)
javax.servlet.resources [2.3.0,2.6.0)
org.osgi.framework (1.4)
org.osgi.framework.hooks.service
org.osgi.service.cm (1.2)
org.osgi.service.discovery (1.0.0)
org.osgi.service.distribution (1.0.0)
org.osgi.service.event (1.1)
org.osgi.service.http (1.2)
org.osgi.service.packageadmin (1.2)
org.osgi.service.startlevel (1.1)

Add... Remove Properties...

Total: 13

Automated Management of Dependencies

Dependency Analysis

Overview | Dependencies | Runtime | Extensions | Extension Points | MANIFEST.MF

- Now eclipse OSGi container is ready for distributed service bundle deployments.

7.4 Greeter Demo

- The demo is based on: http://cxf.apache.org/distributed-osgi-greeter-demo-walkthrough.html

-The demo will go through the following steps:

Step 1: Develop the Greeter Interface bundle in Eclipse.

Step 2: Develop the Greeter Service Implementation bundle in Eclipse.

Step 3: Deploy Greeter Service as web service.

Step 4: Export bundles as jar files.

Step 5: Deploy above jar files in Apache Felix.

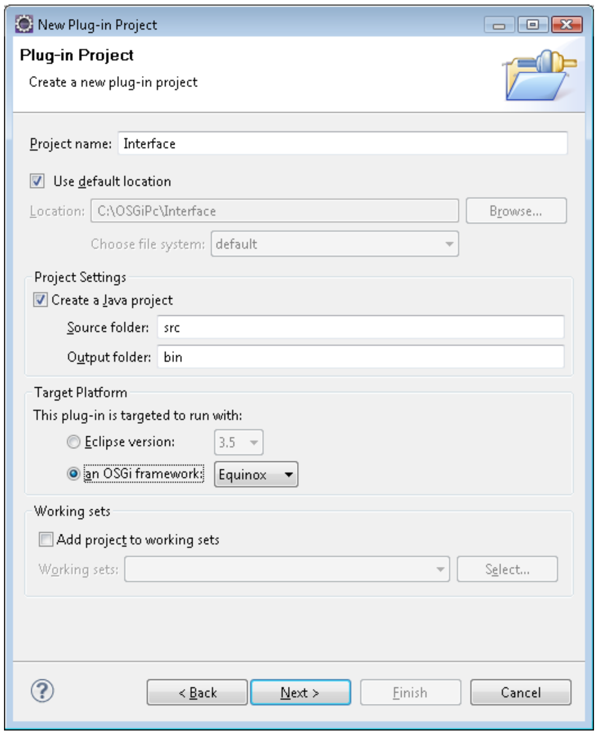
Step 6: Develop and Deploy the Greeter Interface bundle and the Greeter Service Consumer bundle in Eclipse that consume Greeter Service deployed in Apache Felix.

Step 1: Develop the Greeter Interface bundle in Eclipse

- Create new plugin-project called ‘Interface’

- right click on project explorer -> new -> other -> Plug-in Project ->next

- set Project name to ‘Interface’ and select run with OSGi framework Equinox -> next


New Plug-in Project

Plug-in Project

Create a new plug-in project

Project name: Interface

[✓] Use default location

Location: C:\OSGiPc\Interface

Choose file system: default

Project Settings

[✓] Create a Java project

Source folder: src

Output folder: bin

Target Platform

This plug-in is targeted to run with:

( ) Eclipse version: 3.5

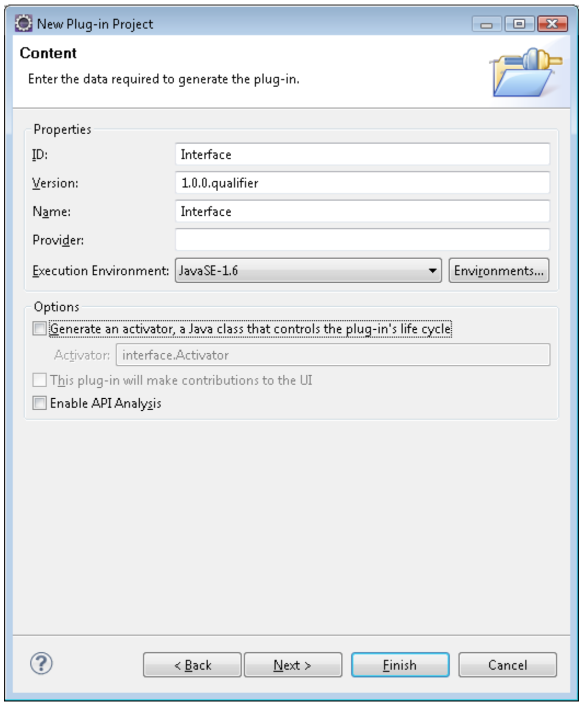
(●) an OSGi framework: Equinox

Working sets

[ ] Add project to working sets

Working sets:

- remove the Activator class->finish


New Plug-in Project

Content

Enter the data required to generate the plug-in.

Properties

ID: Interface

Version: 1.0.0.qualifier

Name: Interface

Provider:

Execution Environment: JavaSE-1.6

Options

[ ] Generate an activator, a Java class that controls the plug-in's life cycle

Activator: interface.Activator

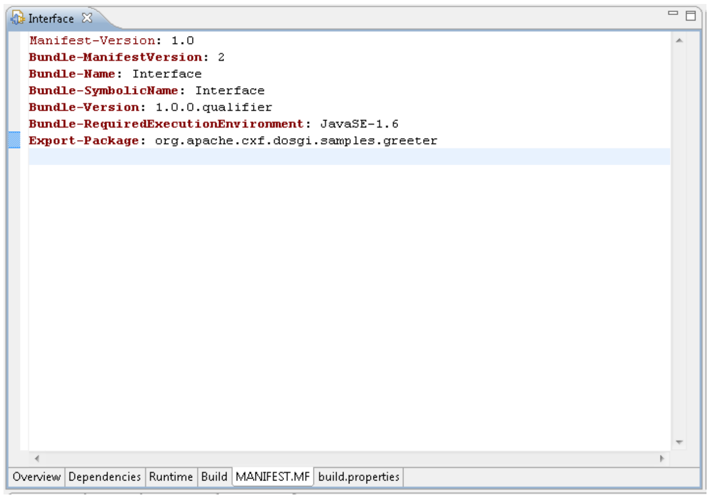
[ ] This plug-in will make contributions to the UI

[ ] Enable API Analysis

- create new package in src called org.apache.cxf.dosgi.samples.greeter

- copy the files GreeterData.java, GreeterException.java, GreeterService.java, GreetingPhrase.java from: http://svn.apache.org/repos/asf/cxf/dosgi/trunk/samples/greeter/interface/src/main/java/org/apache/cxf/dosgi/samples/greeter/

- set import and export packages in MANIFEST.MF as fallows:



```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Interface
Bundle-SymbolicName: Interface
Bundle-Version: 1.0.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Export-Package: org.apache.cxf.dosgi.samples.greeter
```

Step 2: Develop the Greeter Service Implementation bundle in Eclipse.

- create new plugin-project called ‘Impl’

- right click on project explorer -> new -> other -> Plug-in Project ->next

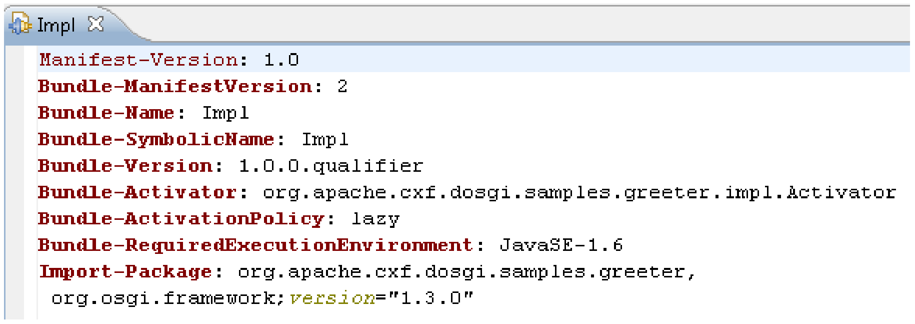
- set Project name to ‘Impl’ and select run with OSGi framework Equinox -> next

- set the activator as: org.apache.cxf.dosgi.samples.greeter.impl.Activator ->finish

- copy files Activator.java, GreeterServiceImpl.java from: http://svn.apache.org/repos/asf/cxf/dosgi/trunk/samples/greeter/impl/src/main/java/org/apache/cxf/dosgi/samples/greeter/impl/

- in the Activator class add org.apache.cxf.dosgi.samples.greeter.GreeterService as imported packages.

- now MANIFEST.MF look like as follows;



Impl

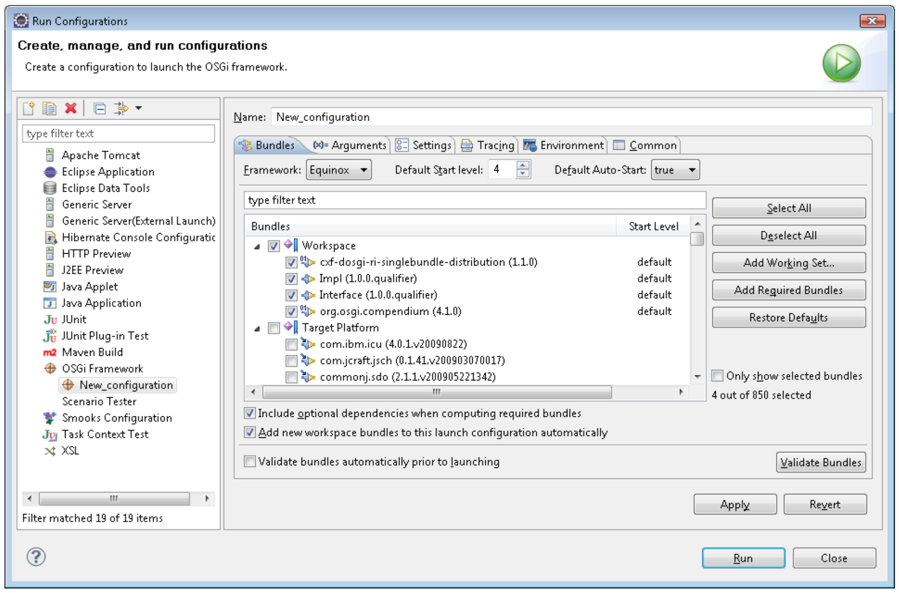
```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Impl
Bundle-SymbolicName: Impl
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: org.apache.cxf.dosgi.samples.greeter.impl.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.apache.cxf.dosgi.samples.greeter,
 org.osgi.framework;version="1.3.0"
```

Step 3: Deploy Greeter Service as web service.

- right click on cxf-dosgi-ri-singlebundle-distribution project -> Run as -> Run Configuration

- create new configuration on OSGi framework name the new configuration.

- select only the workspace bundle as follows:



Run Configurations

Create, manage, and run configurations

Create a configuration to launch the OSGi framework.

Name: New_configuration

Framework: Equinox   Default Start level: 4   Default Auto-Start: true

Bundles	Start Level
[✓] Workspace	
[✓] cxf-dosgi-ri-singlebundle-distribution (1.1.0)	default
[✓] Impl (1.0.0.qualifier)	default
[✓] Interface (1.0.0.qualifier)	default
[✓] org.osgi.compendium (4.1.0)	default
[ ] Target Platform	
[ ] com.ibm.icu (4.0.1.v20090822)	
[ ] com.jcraft.jsch (0.1.41.v200903070017)	
[ ] commonj.sdo (2.1.1.v200905221342)	

[ ] Only show selected bundles

4 out of 850 selected

[✓] Include optional dependencies when computing required bundles

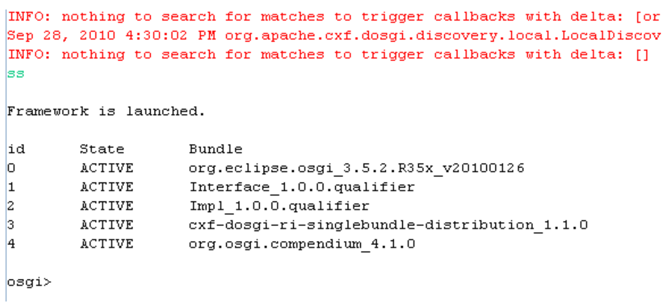
[✓] Add new workspace bundles to this launch configuration automatically

[ ] Validate bundles automatically prior to launching

Filter matched 19 of 19 items

- click Run

- type ss in console to view the status of the bundles:



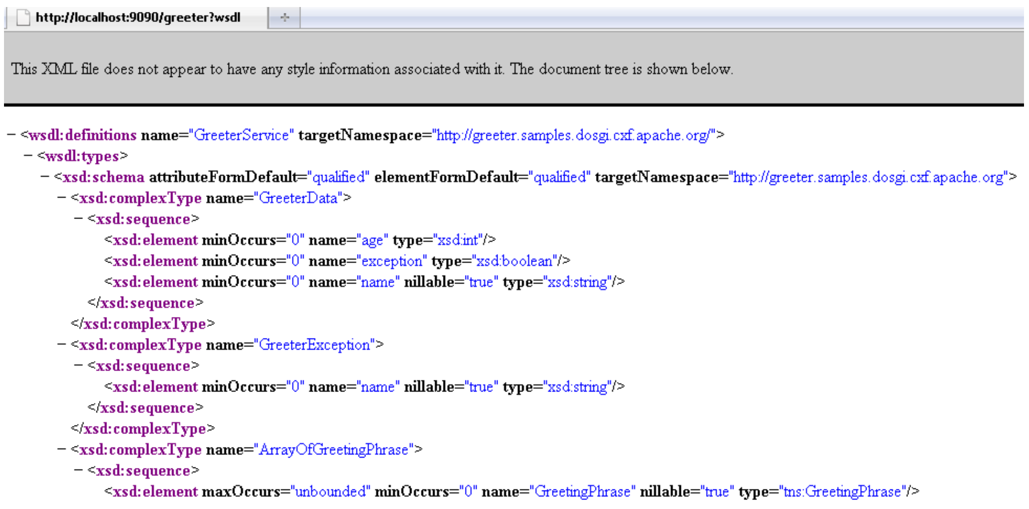
```
INFO: nothing to search for matches to trigger callbacks with delta: [or
Sep 28, 2010 4:30:02 PM org.apache.cxf.dosgi.discovery.local.LocalDiscov
INFO: nothing to search for matches to trigger callbacks with delta: []
ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.5.2.R35x_v20100126
1       ACTIVE      Interface_1.0.0.qualifier
2       ACTIVE      Impl_1.0.0.qualifier
3       ACTIVE      cxf-dosgi-ri-singlebundle-distribution_1.1.0
4       ACTIVE      org.osgi.compendium_4.1.0

osgi>
```

- you can verify running web service by checking http://localhost:9090/greeter?wsdl



http://localhost:9090/greeter?wsdl

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
− <wsdl:definitions name="GreeterService" targetNamespace="http://greeter.samples.dosgi.cxf.apache.org/">
  − <wsdl:types>
    − <xsd:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://greeter.samples.dosgi.cxf.apache.org">
      − <xsd:complexType name="GreeterData">
        − <xsd:sequence>
            <xsd:element minOccurs="0" name="age" type="xsd:int"/>
            <xsd:element minOccurs="0" name="exception" type="xsd:boolean"/>
            <xsd:element minOccurs="0" name="name" nillable="true" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      − <xsd:complexType name="GreeterException">
        − <xsd:sequence>
            <xsd:element minOccurs="0" name="name" nillable="true" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      − <xsd:complexType name="ArrayOfGreetingPhrase">
        − <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0" name="GreetingPhrase" nillable="true" type="tns:GreetingPhrase"/>
```

Step 4: Export bundles as jar files.

- right click on the bundle -> export -> Java -> JAR file -> next

- select the bundle, name and target location

- select options -> next

- select use existing manifest file from the workspace -> finish

- delete Interface, Impl bundles in Eclipse (use ‘close’ exit from osgi prompt)

Step 5: Deploy above jar files in Apache Felix.

-set up felix environment

- move to felix-framework-3.0.2> execute commands:

```
F:\felix-framework-3.0.2>java -jar bin/felix.jar
_______________
Welcome to Apache Felix Gogo

g! install
http://repo1.maven.org/maven2/org/osgi/org.osgi.compendium/4.2.0/org.osgi.com
pendium-4.2.0.jar
Bundle ID: 5
g! start http://www.apache.org/dist/cxf/dosgi/1.2/cxf-dosgi-ri-singlebundle-
distribution-1.2.jar
..
INFO: TopologyManager: triggerExportImportForRemoteSericeAdmin()
g!
```

- start interface, impl bundles in felix:

```
g! start file:/F:/Jar/interface.jar

g! start file:/F:/Jar/impl.jar
```

- to check status of the bundles:

```
lb
START LEVEL 1
ID|State      |Level|Name
0|Active     |    0|System Bundle (3.0.2)
1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
2|Active     |    1|Apache Felix Gogo Command (0.6.0)
3|Active     |    1|Apache Felix Gogo Runtime (0.6.0)
4|Active     |    1|Apache Felix Gogo Shell (0.6.0)
5|Resolved   |    1|osgi.cmpn (4.2.0.200908310645)
6|Active     |    1|Distributed OSGi Distribution Software Single-Bundle Dis-
tribution (1.2.0)
7|Active     |    1|Interface (1.0.0.qualifier)
8|Active     |    1|Impl (1.0.0.qualifier)
g!
```

- you can verify running web service by checking http://localhost:9090/greeter?wsdl

Step 6: Develop and Deploy the Greeter Interface bundle and the Greeter Service Consumer bundle in Eclipse

- develop Greeter Interface bundle similar to above.

- create new plugin project called ‘client’ and set Activator class as:

org.apache.cxf.dosgi.samples.greeter.client.Activator

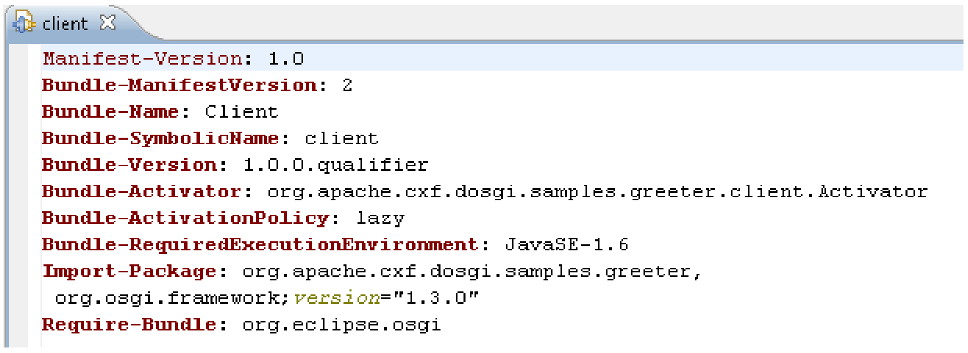
- copy Activator.java, GreeterDataImpl.java, GreeterDialog.java to src from:

http://svn.apache.org/repos/asf/cxf/dosgi/trunk/samples/greeter/client/src/main/java/org/apache/cxf/dosgi/samples/greeter/client/

- add org.apache.cxf.dosgi.samples.greeter as imported package.

- add org.osgi.framework as imported package

- manifest file looks like as fallows



```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Client
Bundle-SymbolicName: client
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: org.apache.cxf.dosgi.samples.greeter.client.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.apache.cxf.dosgi.samples.greeter,
 org.osgi.framework;version="1.3.0"
Require-Bundle: org.eclipse.osgi
```

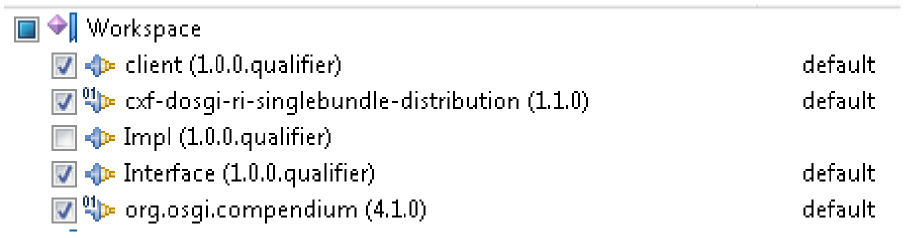
- create remote-services.xml in OSGI-INF\remote-service

```
<?xml version="1.0" encoding="UTF-8"?>
<service-descriptions xmlns="http://www.osgi.org/xmlns/sd/v1.0.0">
  <service-description>
       <provide inter-
face="org.apache.cxf.dosgi.samples.greeter.GreeterService" />
       <property name="osgi.remote.interfaces">*</property>
       <property name="osgi.remote.configuration.type">pojo</property>
       <property
name="osgi.remote.configuration.pojo.address">http://localhost:9090/greeter</
property>
 </service-description>

 <!-- further service-description tags are allowed here -->
</service-descriptions>
```

- deploy the client

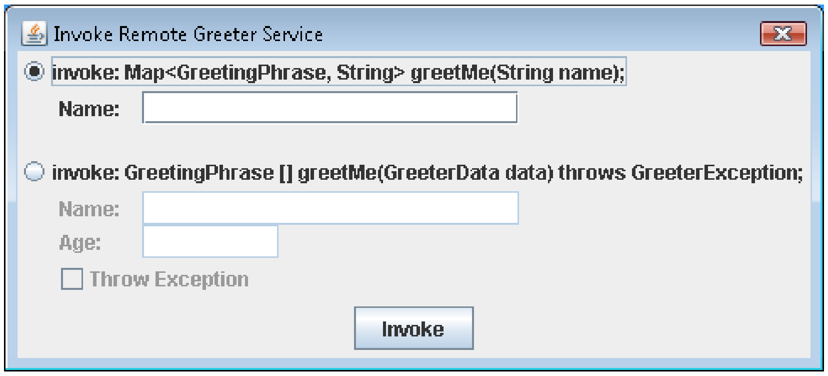
- right click on client -> Run as -> Run Configuration -> select



Workspace	
☑ client (1.0.0.qualifier)	default
☑ cxf-dosgi-ri-singlebundle-distribution (1.1.0)	default
☐ Impl (1.0.0.qualifier)	
☑ Interface (1.0.0.qualifier)	default
☑ org.osgi.compendium (4.1.0)	default

- click Run

- then client will appear



Invoke Remote Greeter Service

(●) invoke: Map<GreetingPhrase, String> greetMe(String name);

Name:

( ) invoke: GreetingPhrase [] greetMe(GreeterData data) throws GreeterException;

Name:

Age:

[ ] Throw Exception

- output will look like this:

```
*** Opening greeter client dialog ***
*** Invoking greeter ***
greetMe("Cool") returns:
        Hola Cool
        Bonjour Cool
        Hoi Cool
        Hello Cool
*** Opening greeter client dialog ***
```